# Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall and Winter 2025
Lecture 11

# Local Search

- Vertex cover
- Hopfield neural networks
- Max-cut
- Approximation algorithm for vertex cover
- Take-home messages

# Local Search

- Vertex cover
- Hopfield neural networks
- Max-cut
- Approximation algorithm for vertex cover
- Take-home messages

**Solve problems *approximately***

**Solve problems *approximately***

**—— aims at a local optimum**

**Solve problems *approximately***

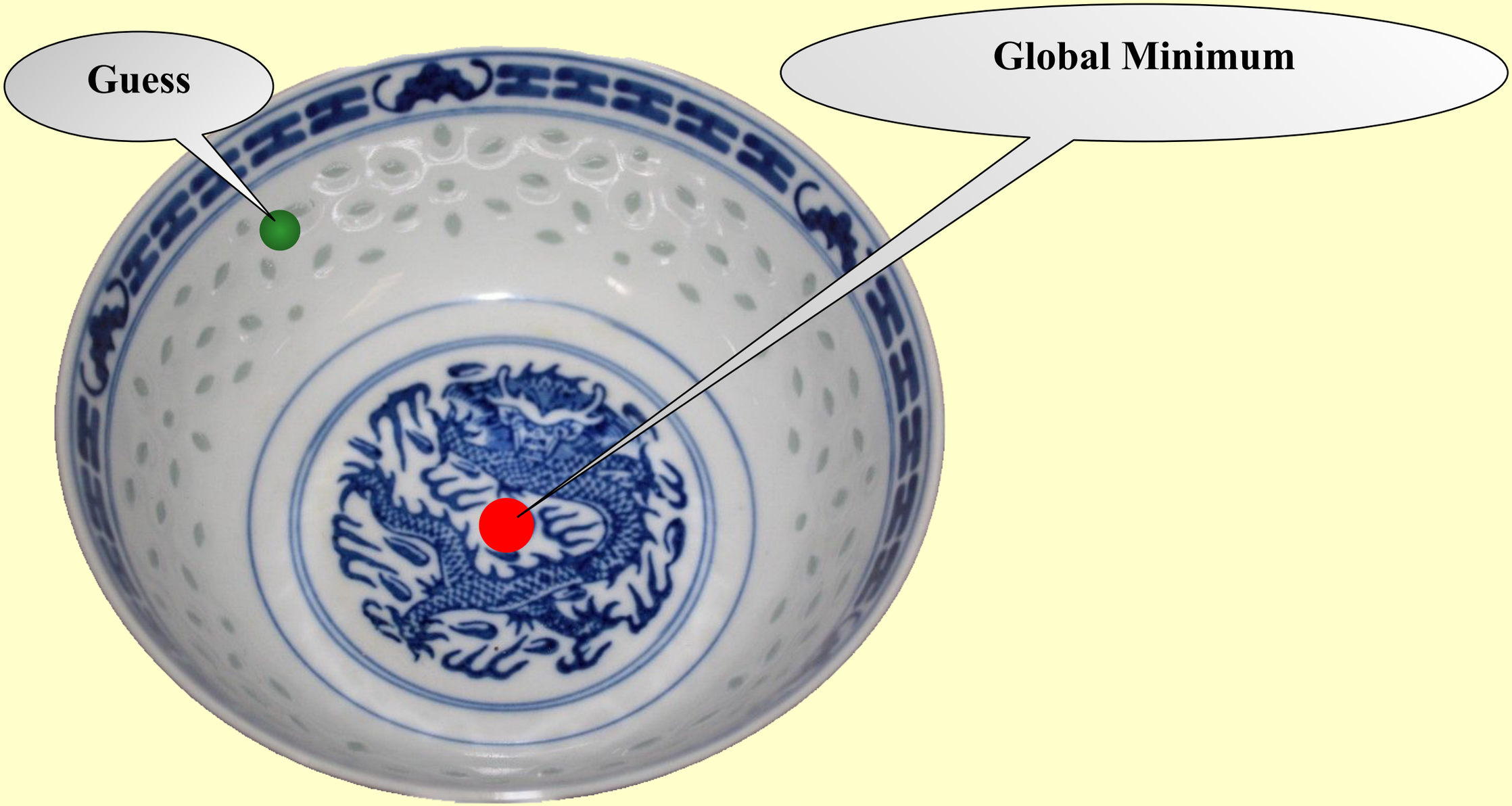
**—— aims at a local optimum**

**Solve problems *approximately***

**—— aims at a local optimum**

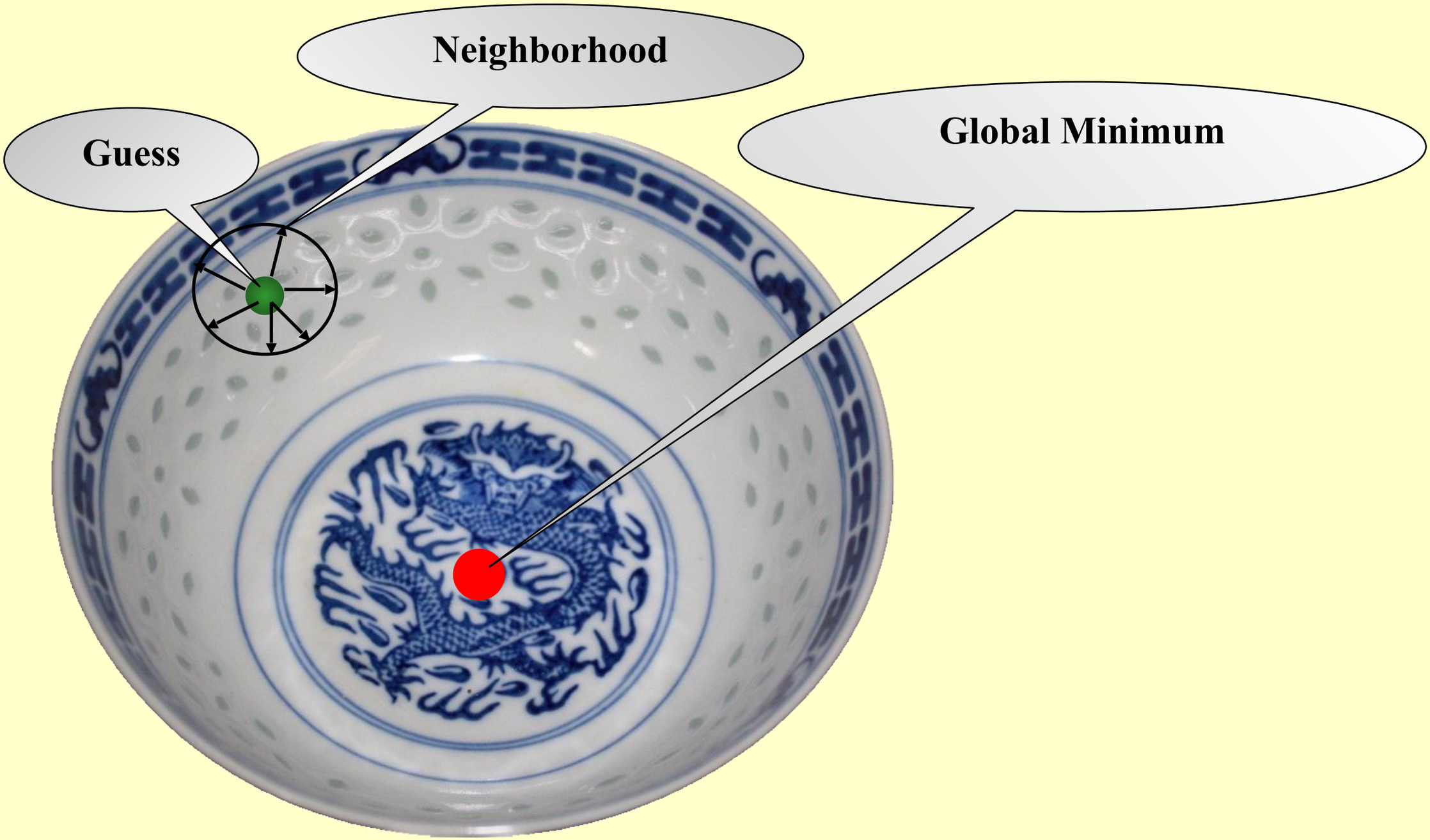
**Solve problems *approximately***

**—— aims at a local optimum**



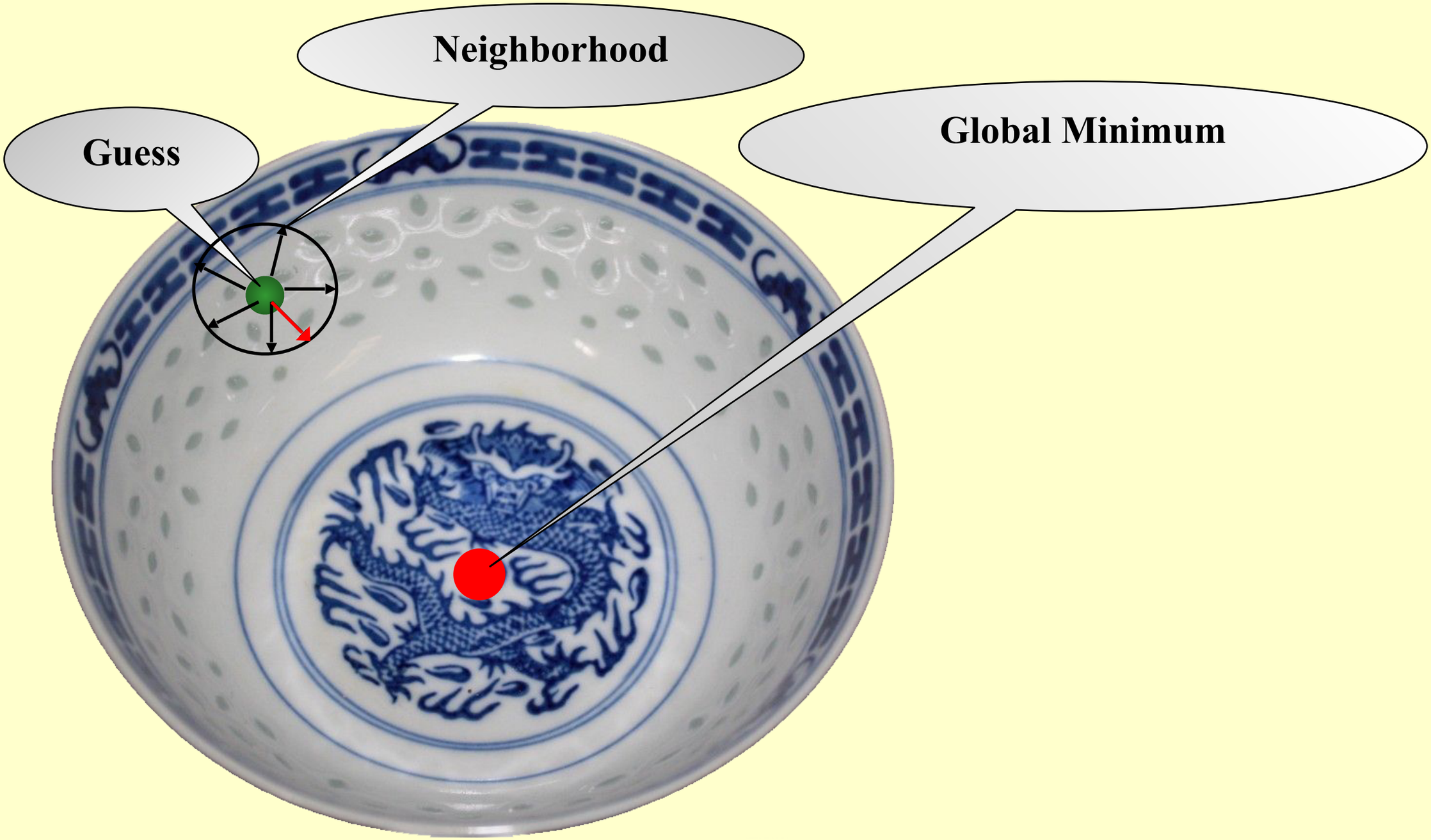
**Solve problems *approximately***

**—— aims at a local optimum**



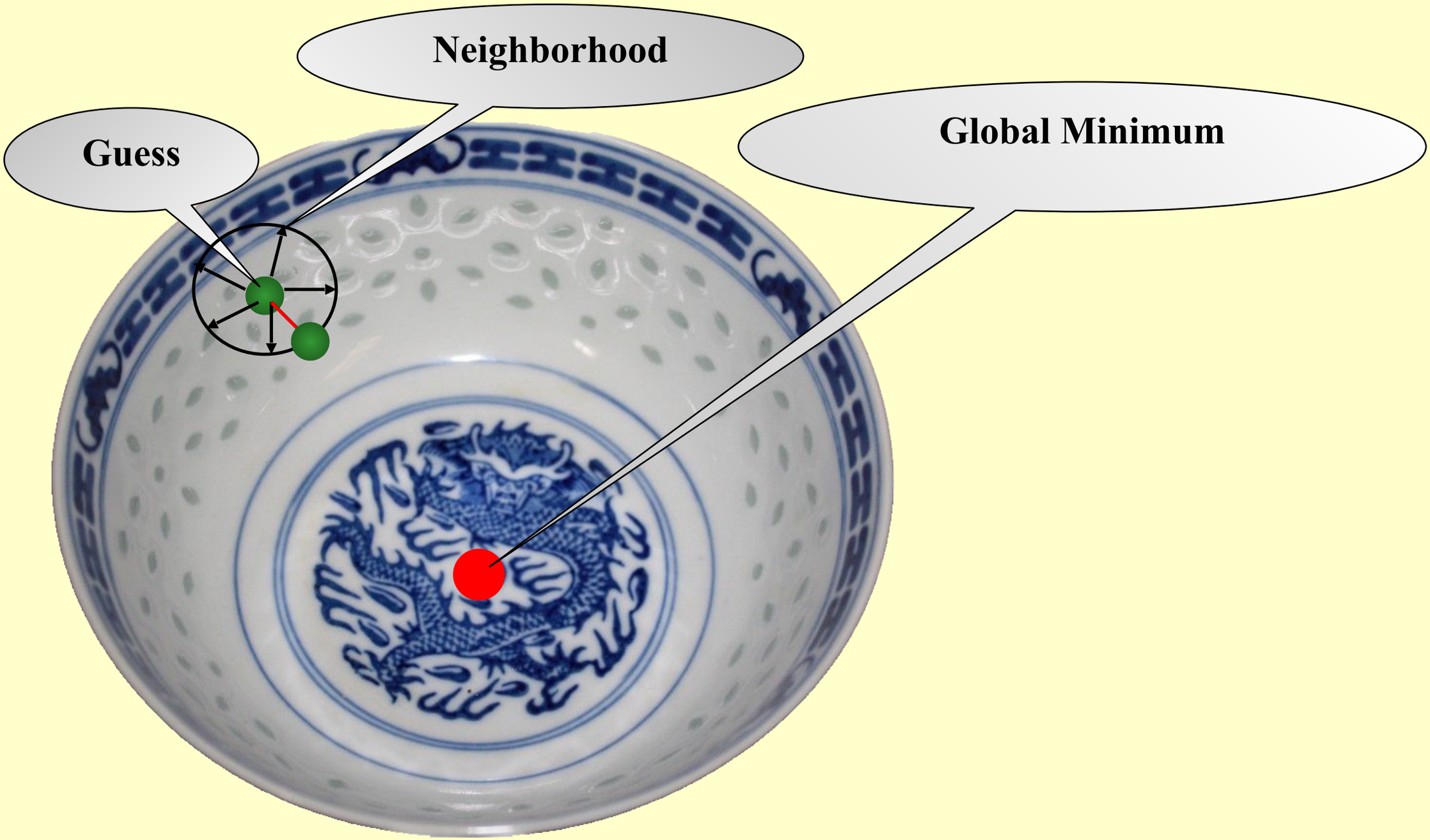
**Solve problems *approximately***

**—— aims at a local optimum**



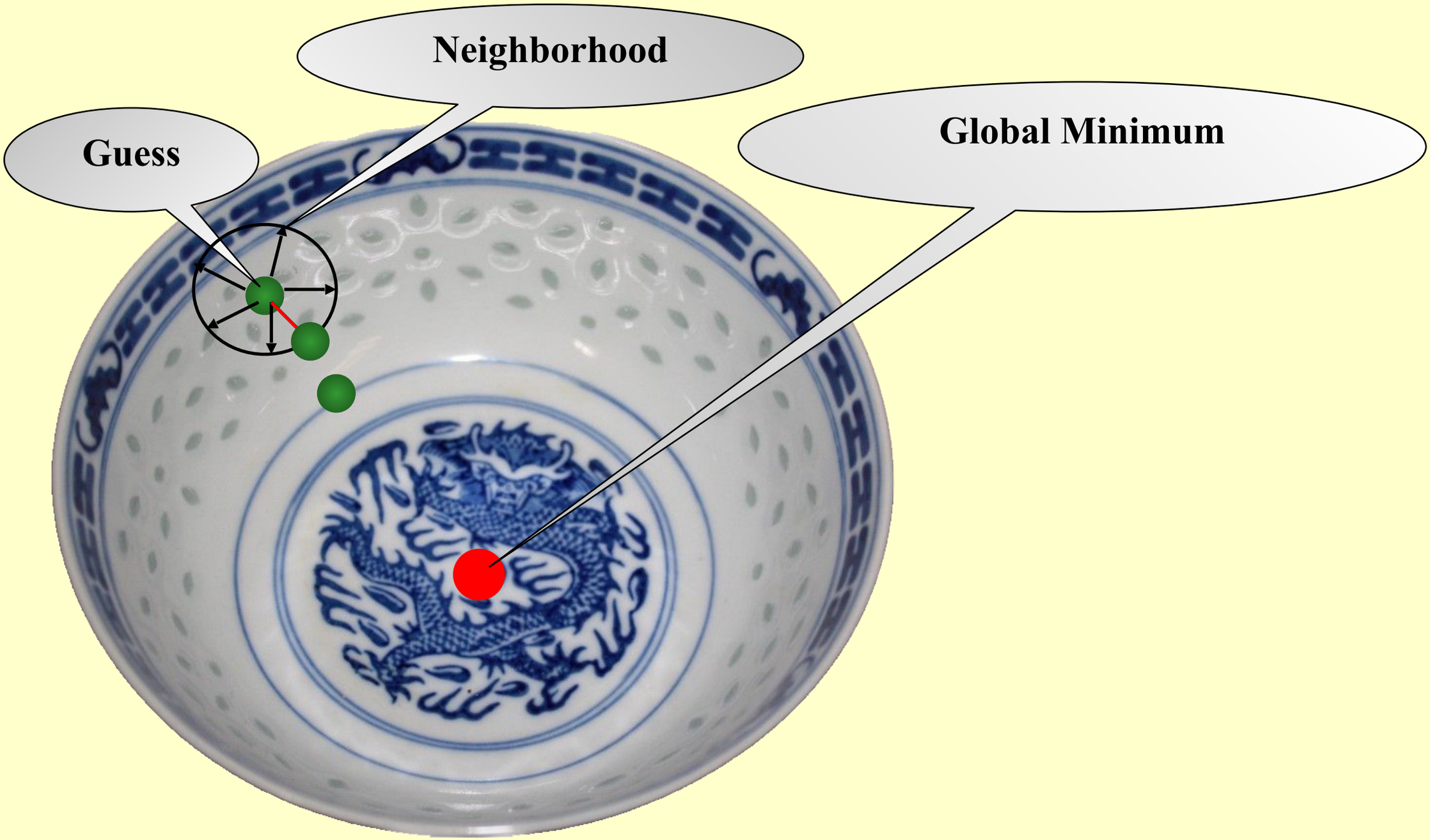
**Solve problems *approximately***

**—— aims at a local optimum**



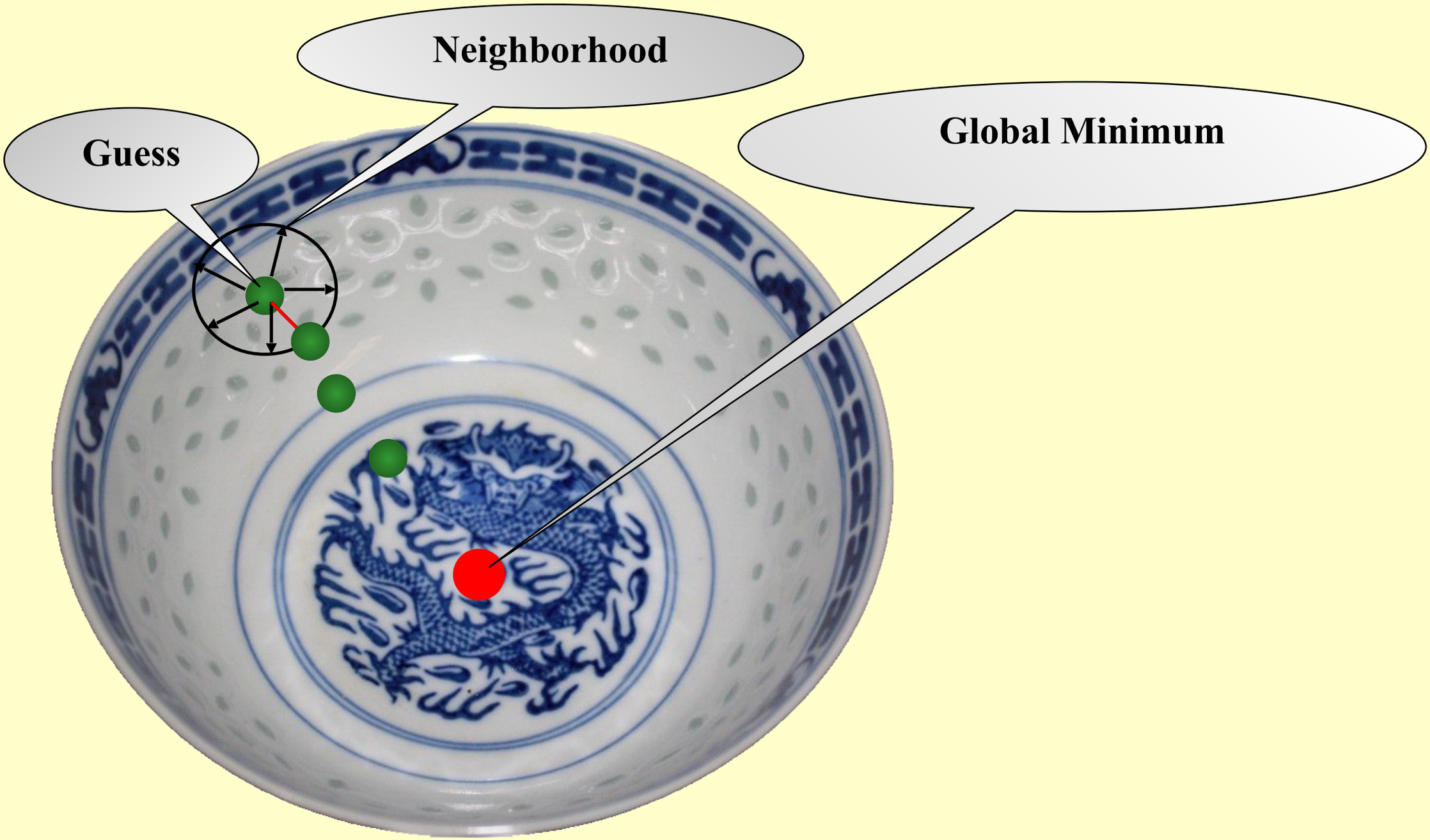
**Solve problems *approximately***

**—— aims at a local optimum**



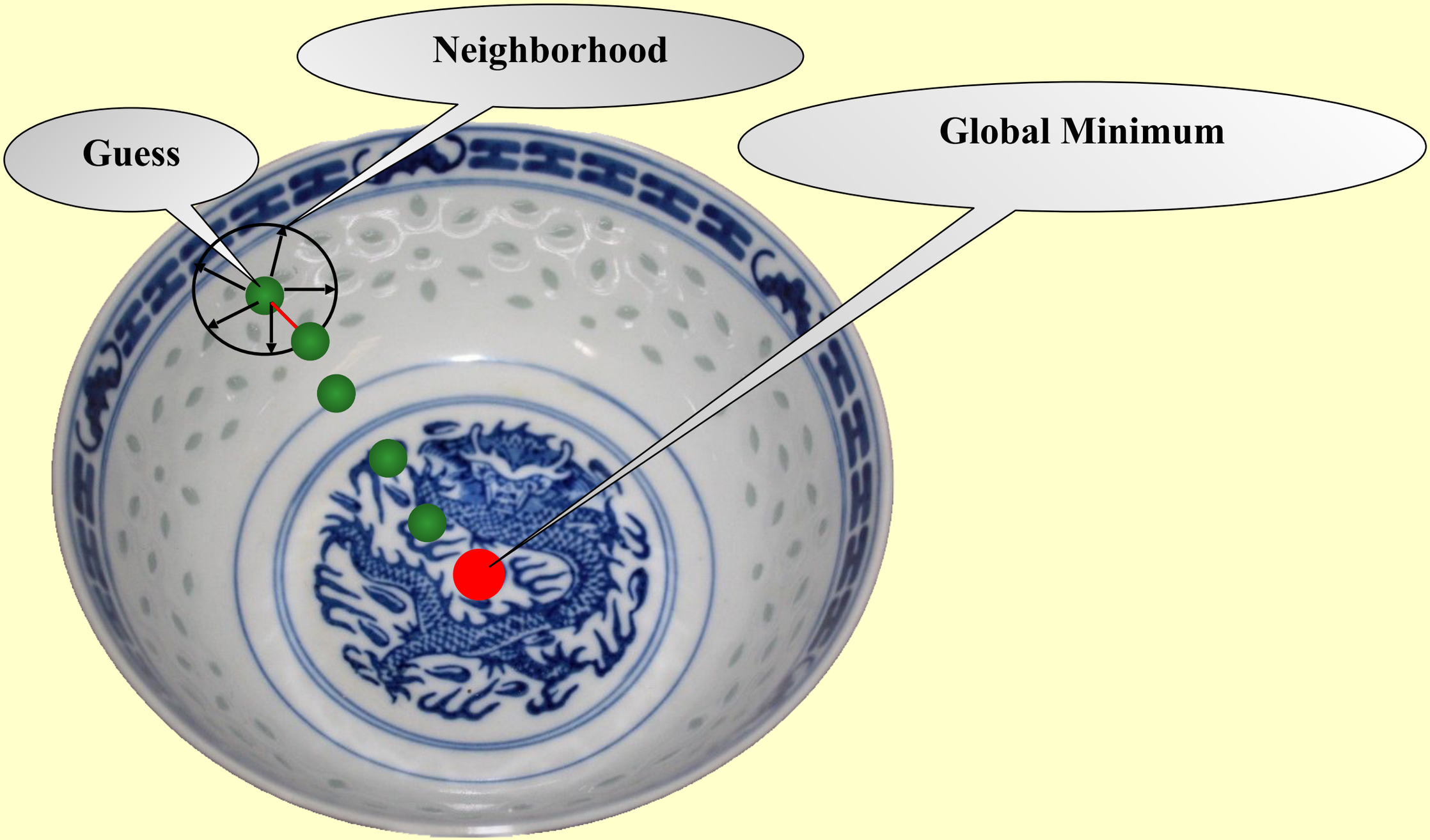
**Solve problems *approximately***

**—— aims at a local optimum**



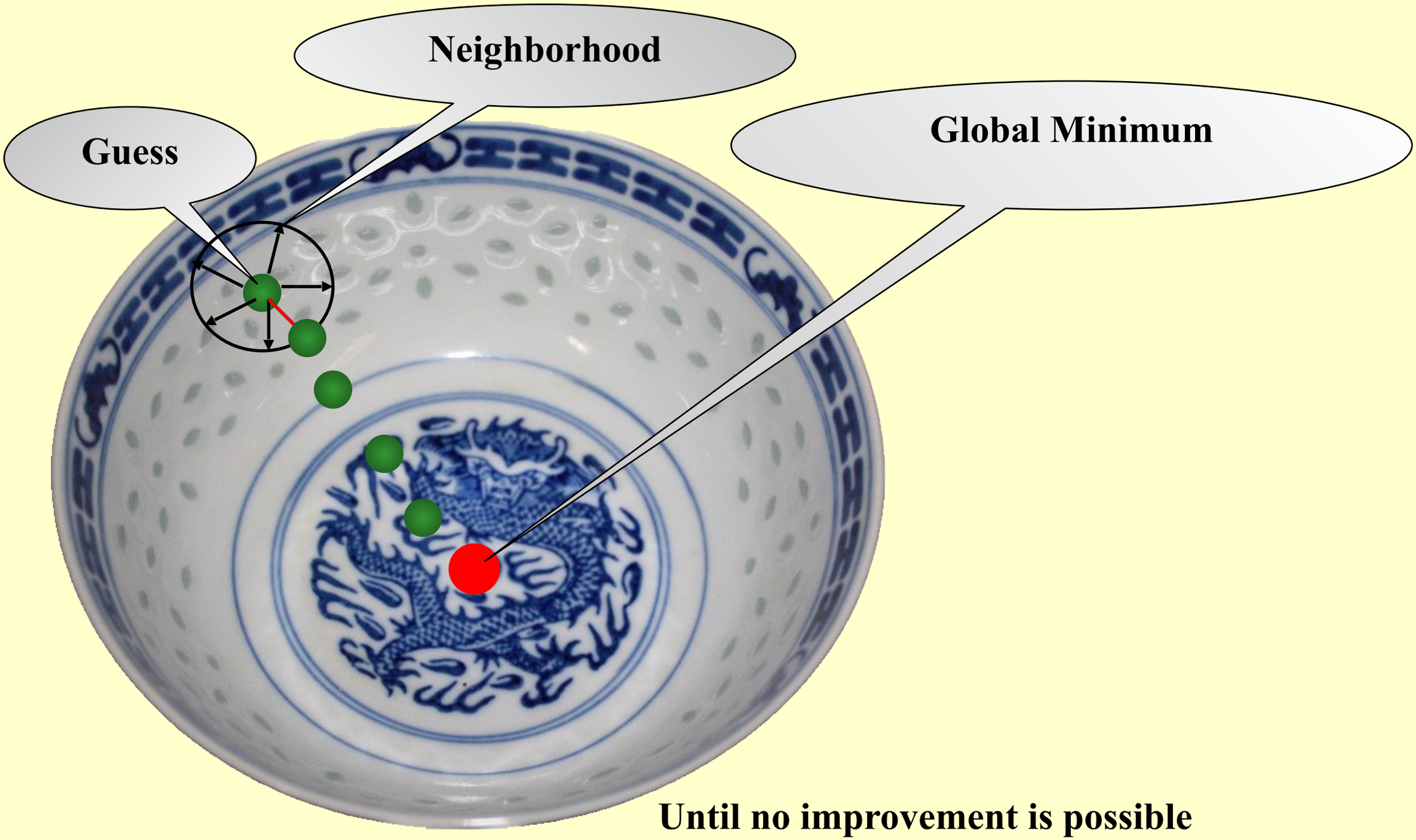
**Solve problems *approximately***

**—— aims at a local optimum**



**Solve problems *approximately***

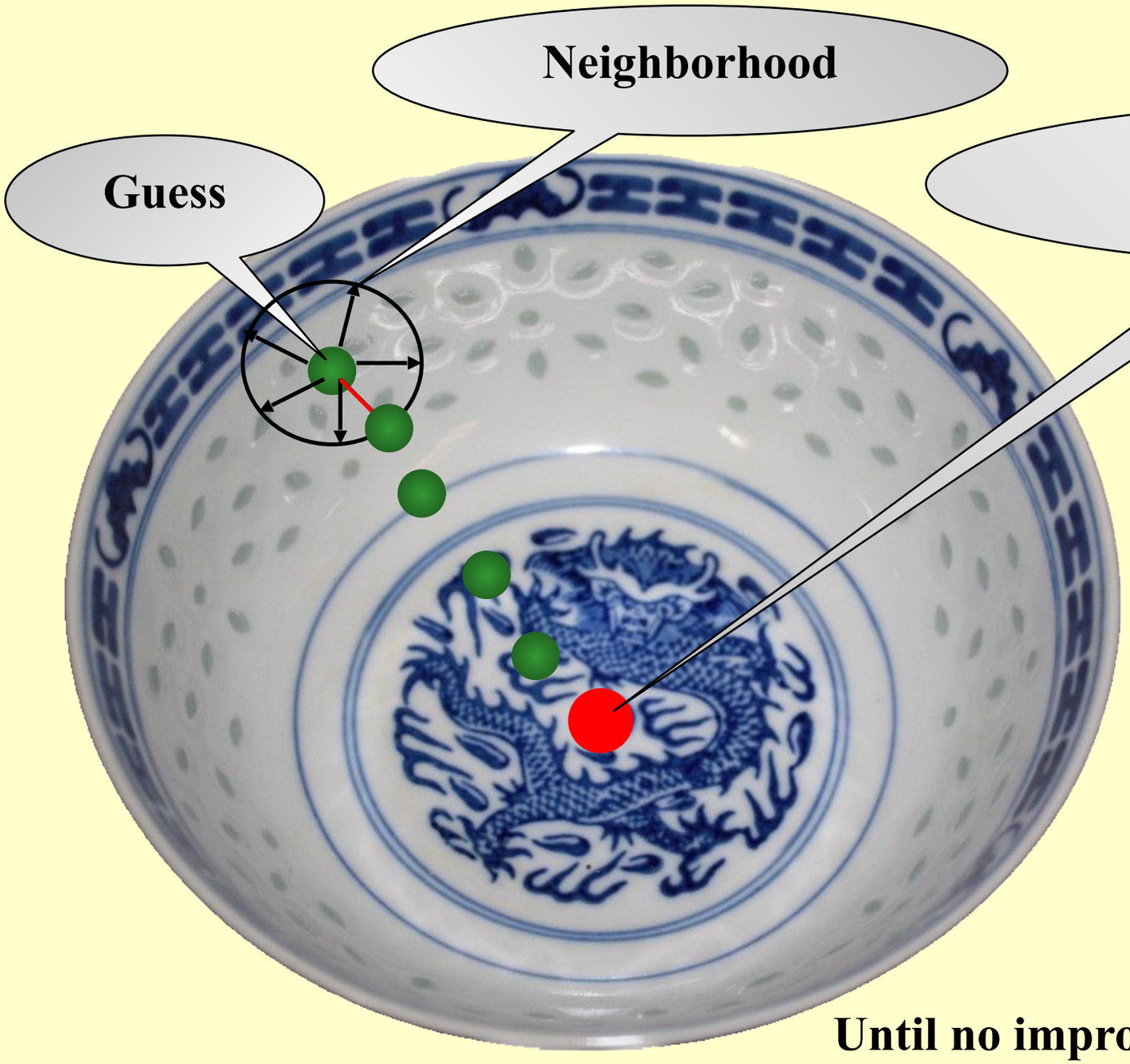
**—— aims at a local optimum**

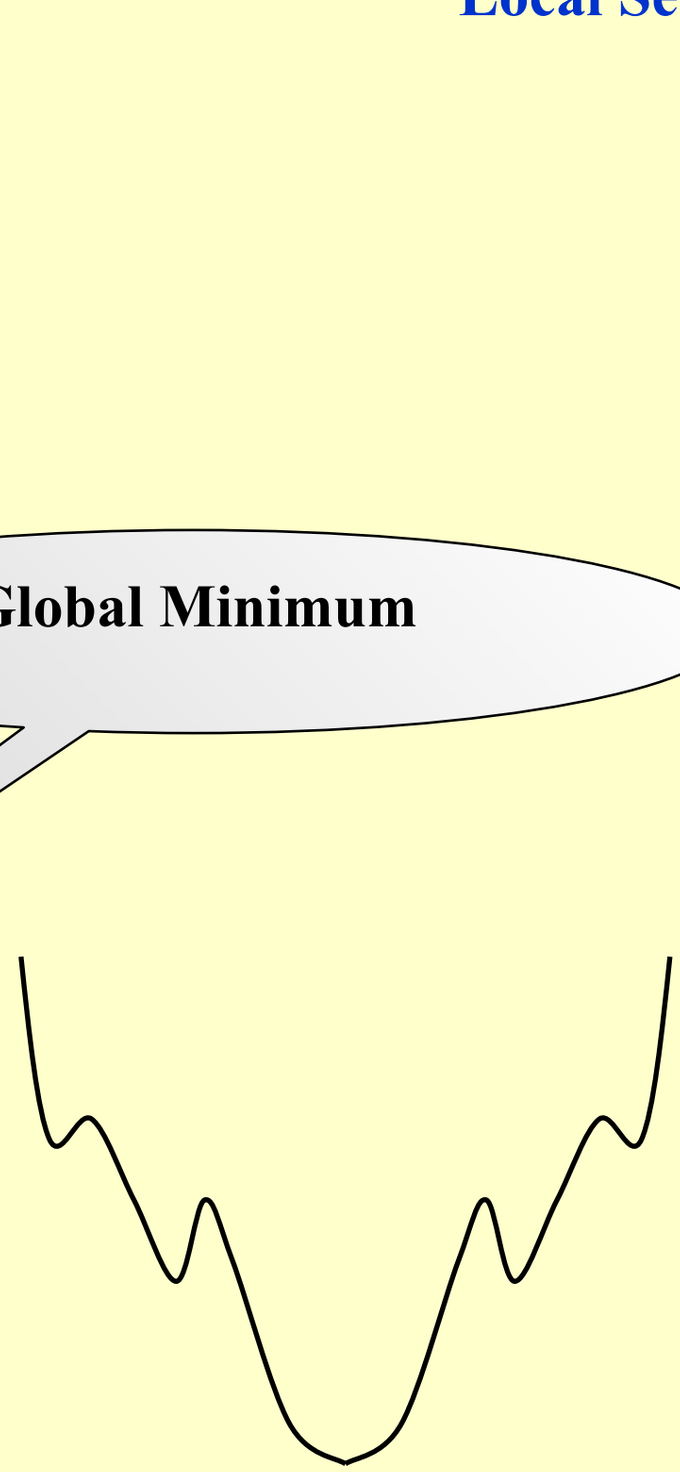



**Until no improvement is possible**

**Solve problems *approximately***

**—— aims at a local optimum**

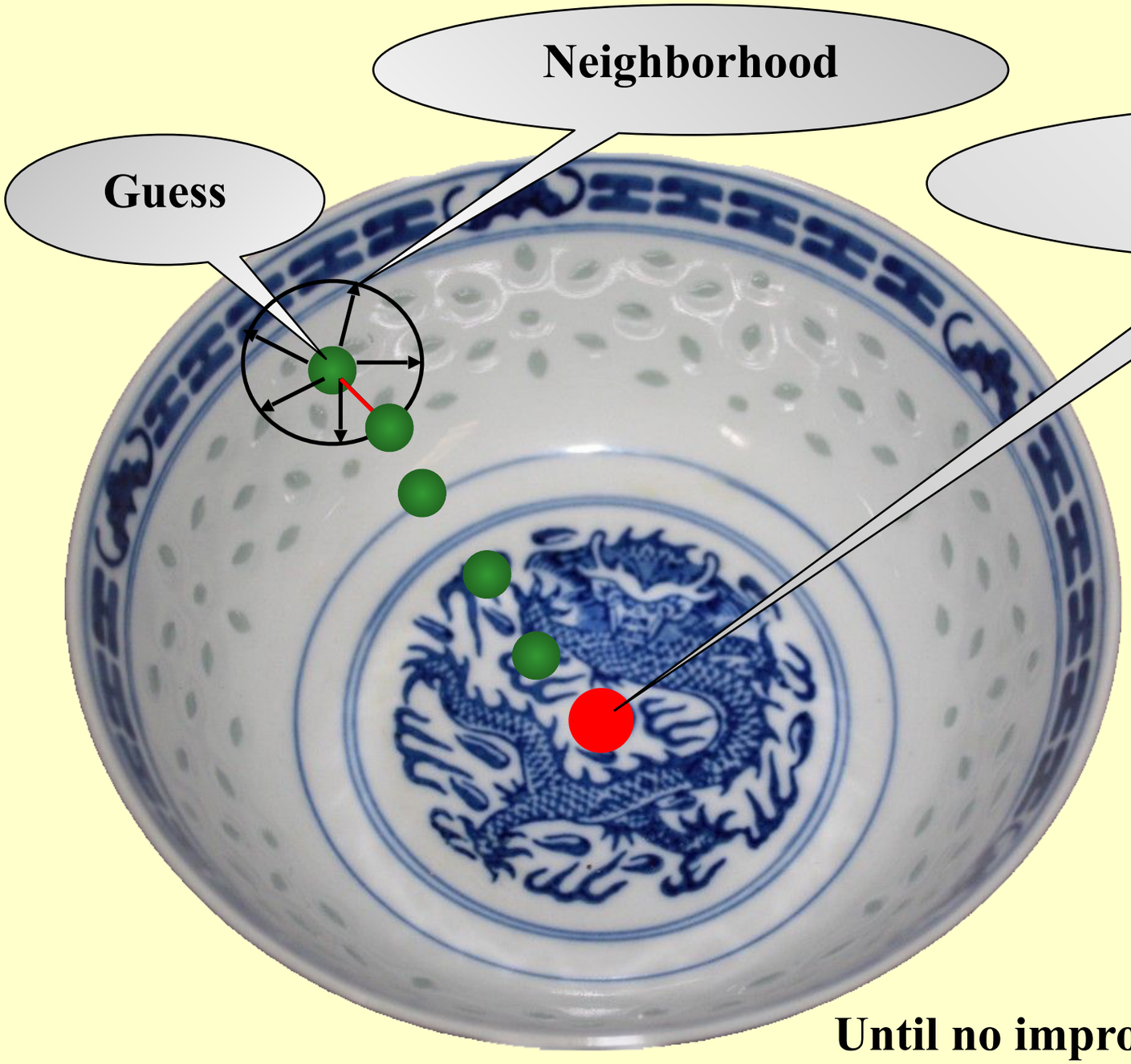


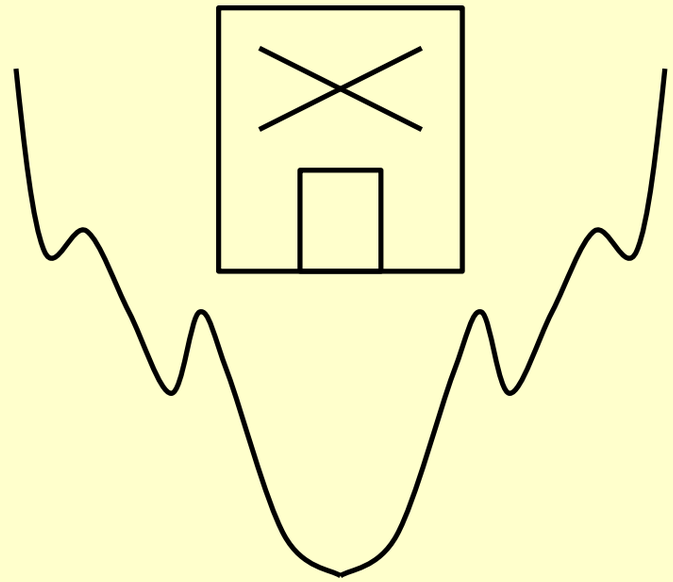



**Until no improvement is possible**

**Solve problems *approximately***

**—— aims at a local optimum**

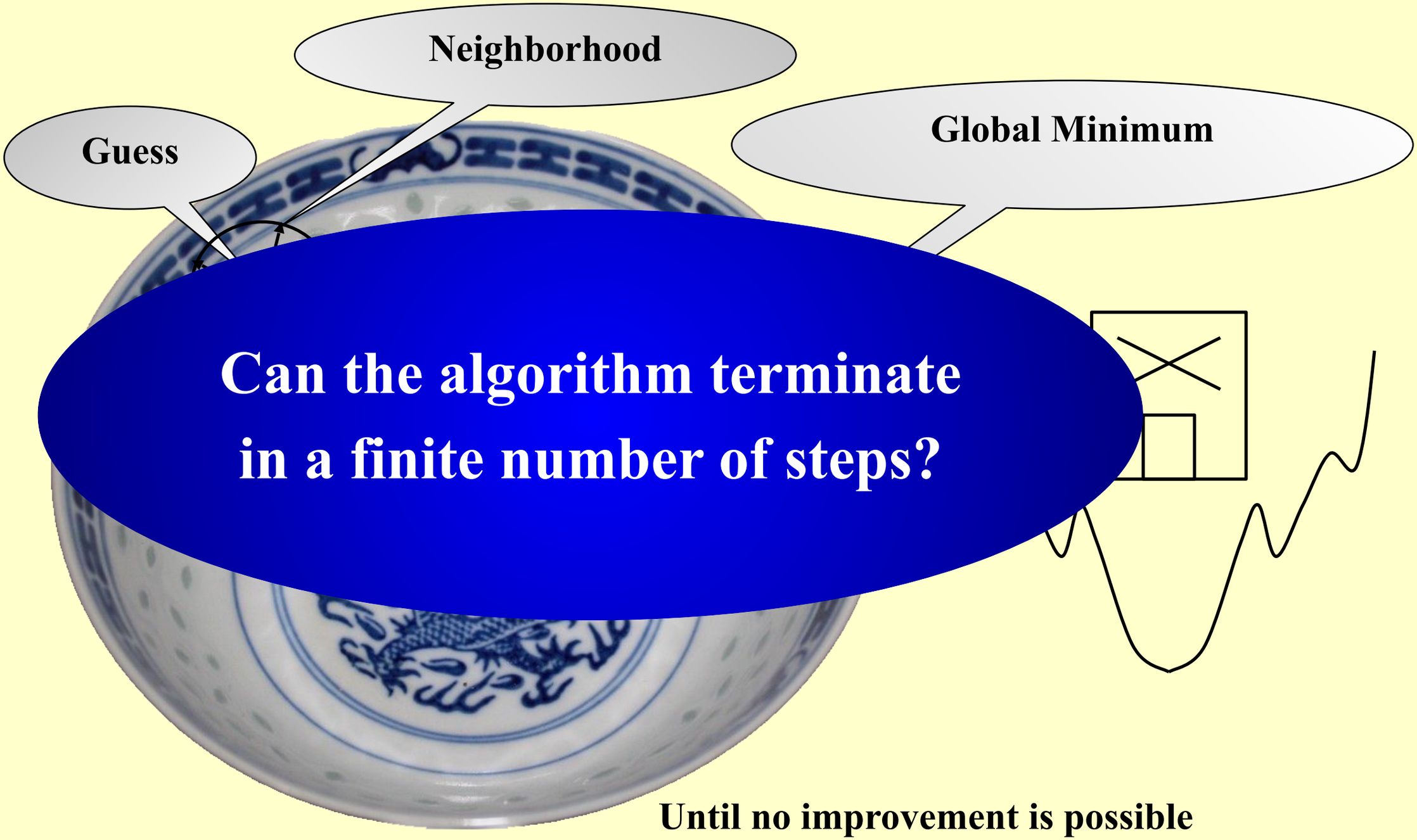


**Until no improvement is possible**

**Solve problems *approximately***

**—— aims at a local optimum**




**Until no improvement is possible**

# Framework of Local Search

# Framework of Local Search

☞ **Local**

# Framework of Local Search

☞ **Local**

☞ **Search**

## Framework of Local Search

☞ **Local**

- **Define *neighborhoods* in the feasible set**
- **A local optimum is a best solution in a neighborhood**

☞ **Search**

## Framework of Local Search

☞ **Local**

- **Define *neighborhoods* in the feasible set**
- **A local optimum is a best solution in a neighborhood**

☞ **Search**

- **Start with a feasible solution and search a better one within the neighborhood**
- **A local optimum is achieved if no improvement is possible**

## Neighbor Relation

## Neighbor Relation

☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**

## Neighbor Relation

- ☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**
- ☞ **N(S): *neighborhood* of S – the set { S': S ~ S' }.**

## Neighbor Relation

☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**

☞ **N(S): *neighborhood* of S – the set { S': S ~ S' }.**

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

## Neighbor Relation

☞ S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.

☞ N(S): *neighborhood* of S – the set { S': S ~ S' }.

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

## Neighbor Relation

☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**

☞ **N(S): *neighborhood* of S – the set { S': S ~ S' }.**

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

〖Example〗 The Vertex Cover Problem.

〖Example〗 The Vertex Cover Problem.

❖ **Vertex cover problem:** Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

☞ S ~ S':

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$**

☞ **cost(S) = | S |**

☞ **S ~ S': S' can be obtained from S by (adding or)**

***deleting a single node*.**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

☞ S ~ S': S' can be obtained from S by (adding or) *deleting a single node*.

Each vertex cover S has at most |V| neighbors.

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$ : all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

☞ S ~ S': S' can be obtained from S by (adding or)

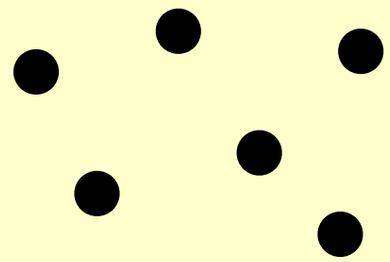
*deleting a single node*.

Each vertex cover S has at most |V| neighbors.

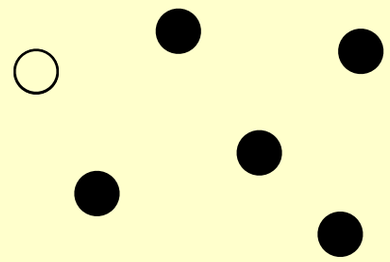
☞ Search: Start from S = V; delete a node and check if S' is a vertex cover with a smaller cost.

**Case 0:**

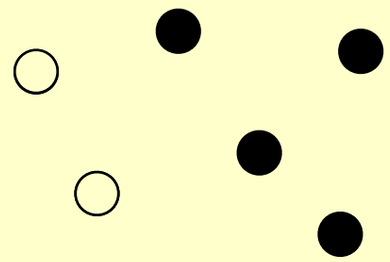
**Case 0:**



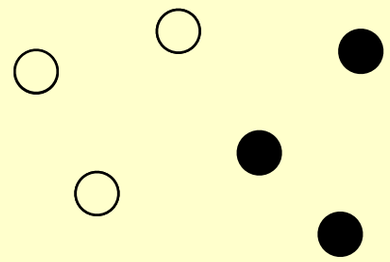
**Case 0:**



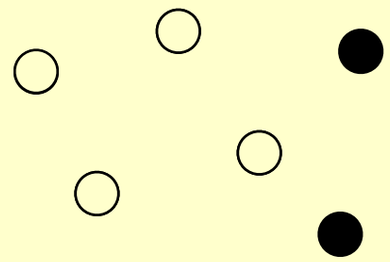
**Case 0:**



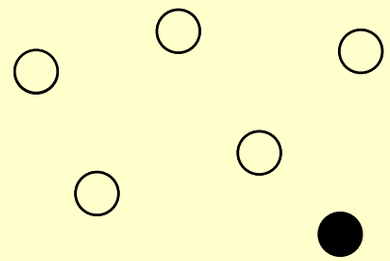
**Case 0:**



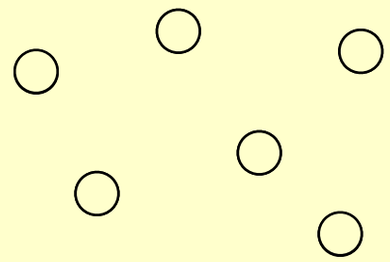
**Case 0:**



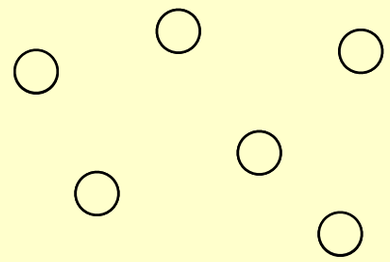
**Case 0:**

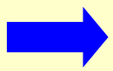


**Case 0:**



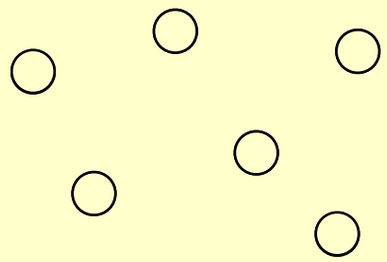
**Case 0:**





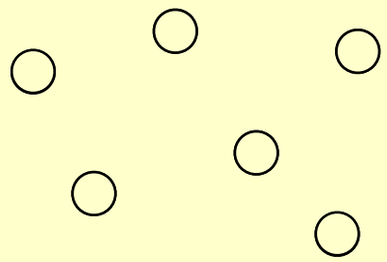
**S = ∅**

**Case 0:**

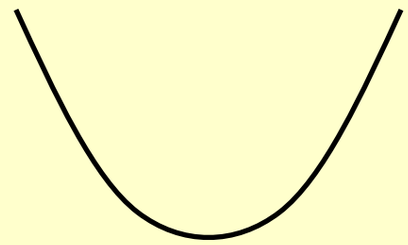


**S = ∅**

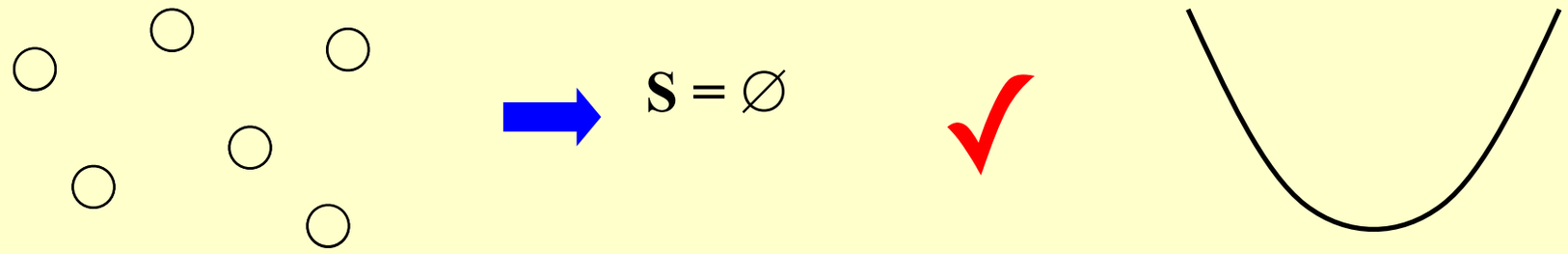
**Case 0:**



$\mathbf{S} = \varnothing$

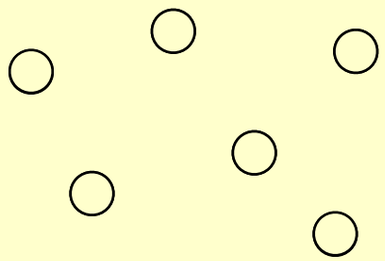


**Case 0:**

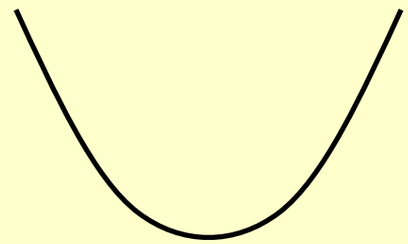



**Case 1:**

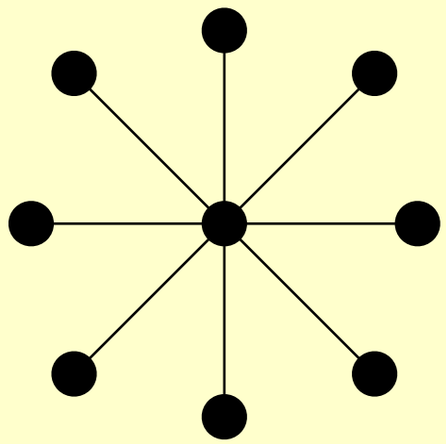
**Case 0:**



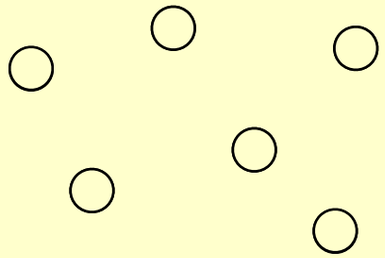
$S = \varnothing$



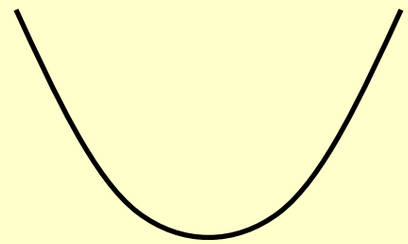
**Case 1:**



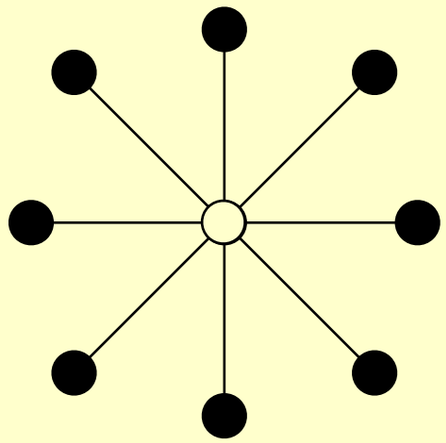
**Case 0:**



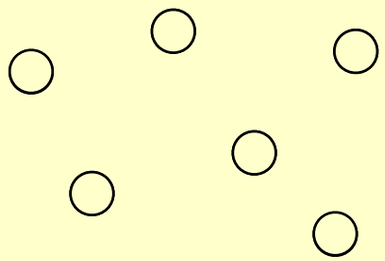
$S = \varnothing$



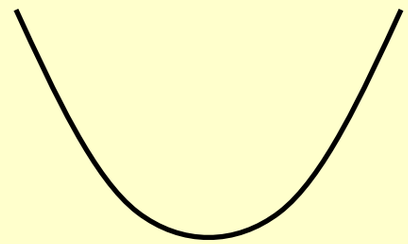
**Case 1:**



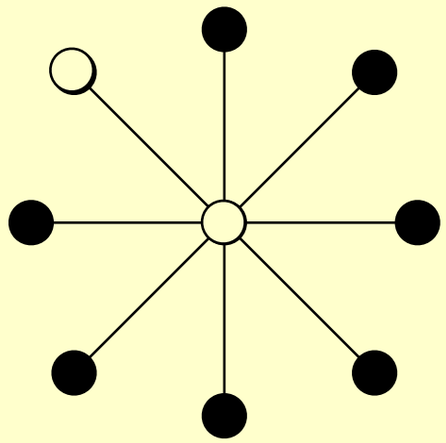
**Case 0:**



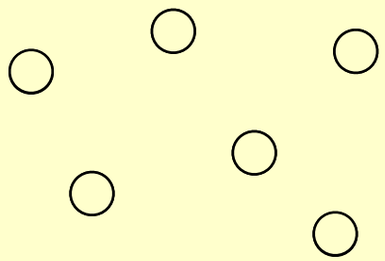
$S = \varnothing$



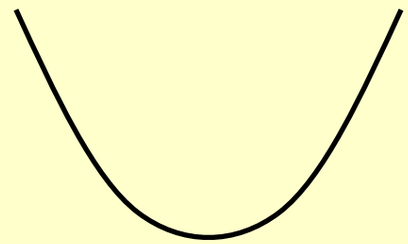
**Case 1:**



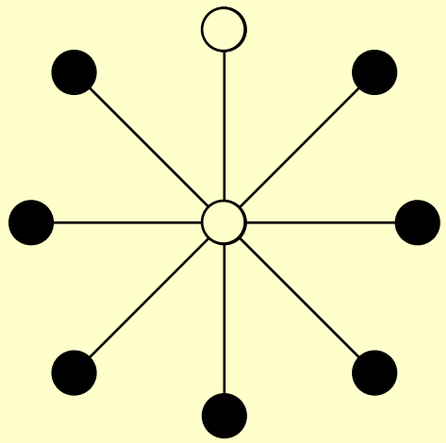
**Case 0:**



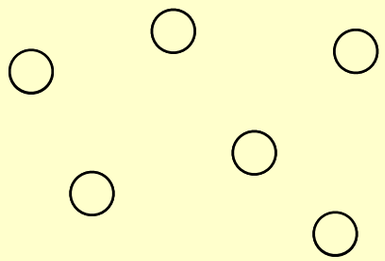
$S = \varnothing$



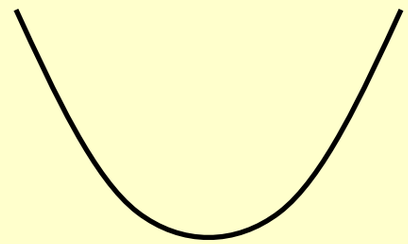
**Case 1:**



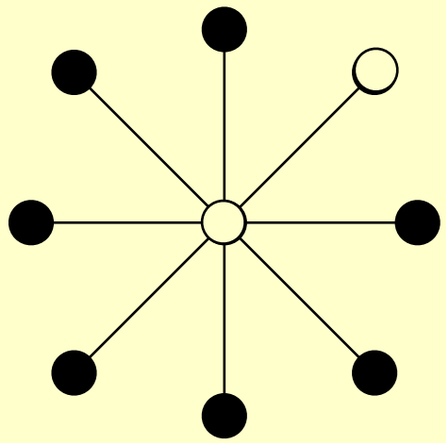
**Case 0:**



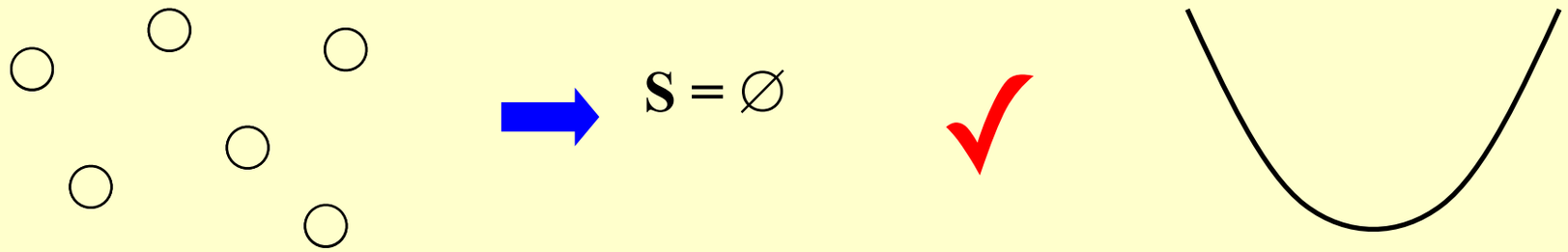
$S = \varnothing$



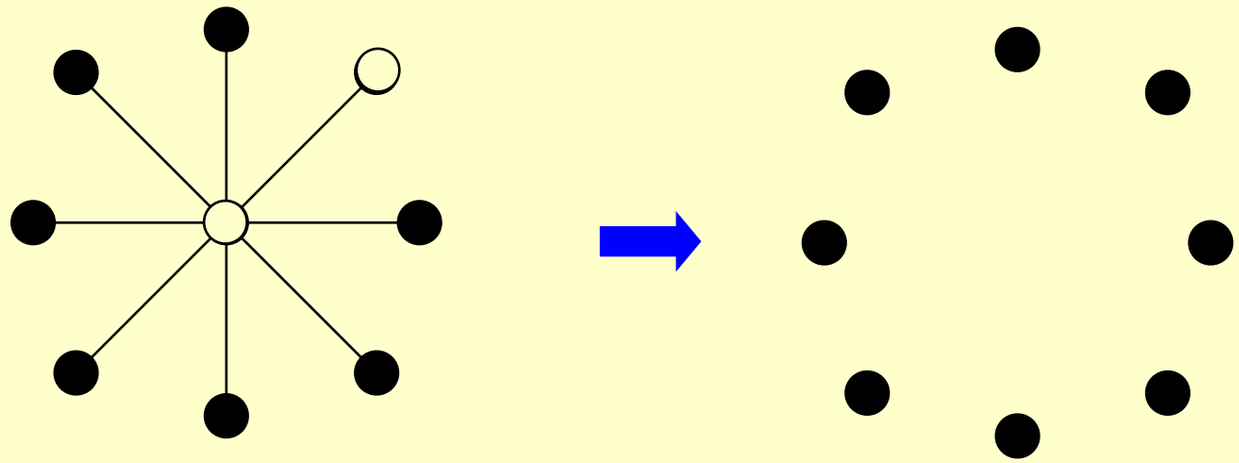
**Case 1:**



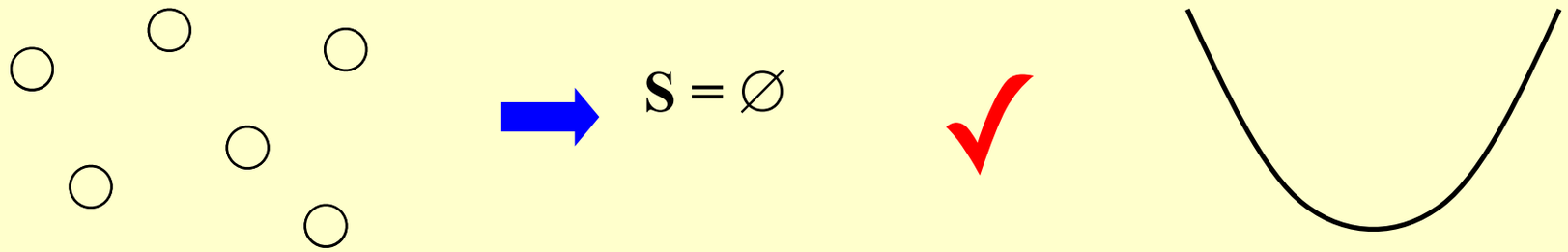
**Case 0:**



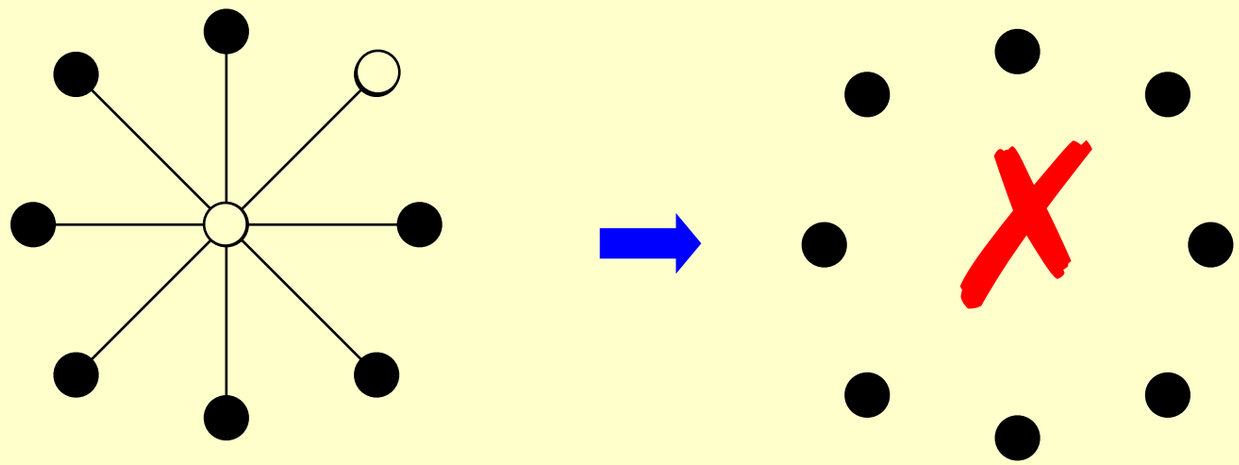

**Case 1:**



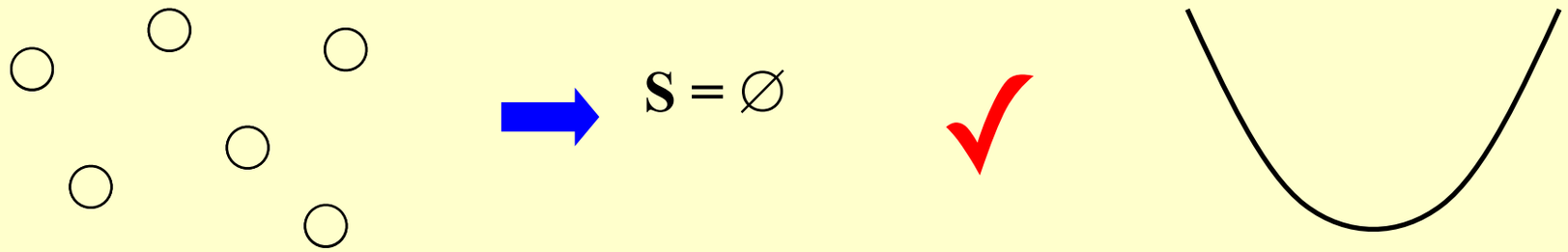
**Case 0:**



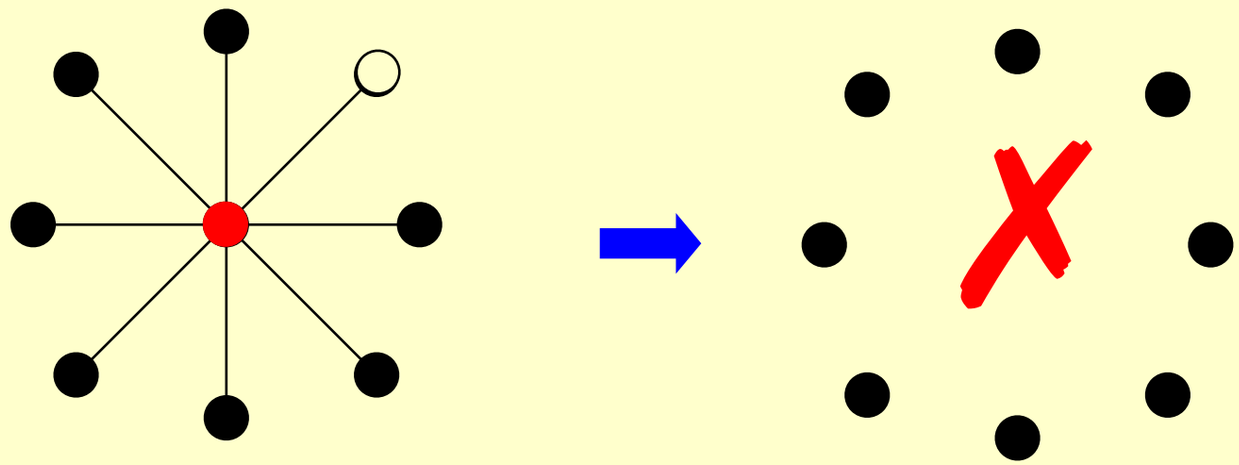

**Case 1:**



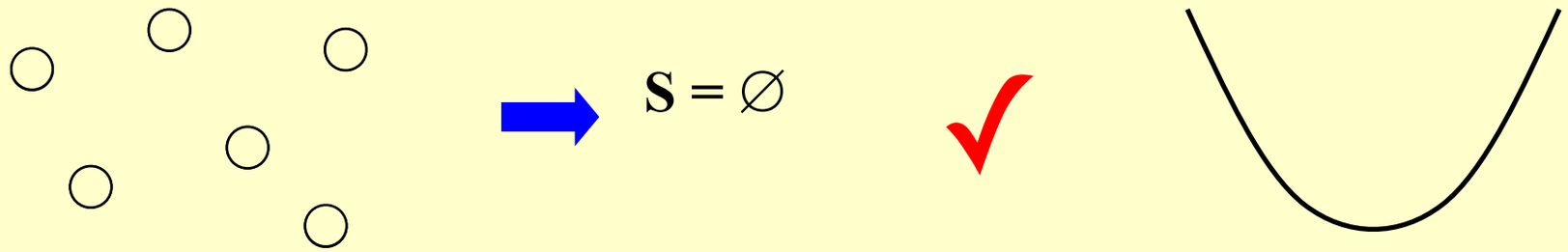
**Case 0:**



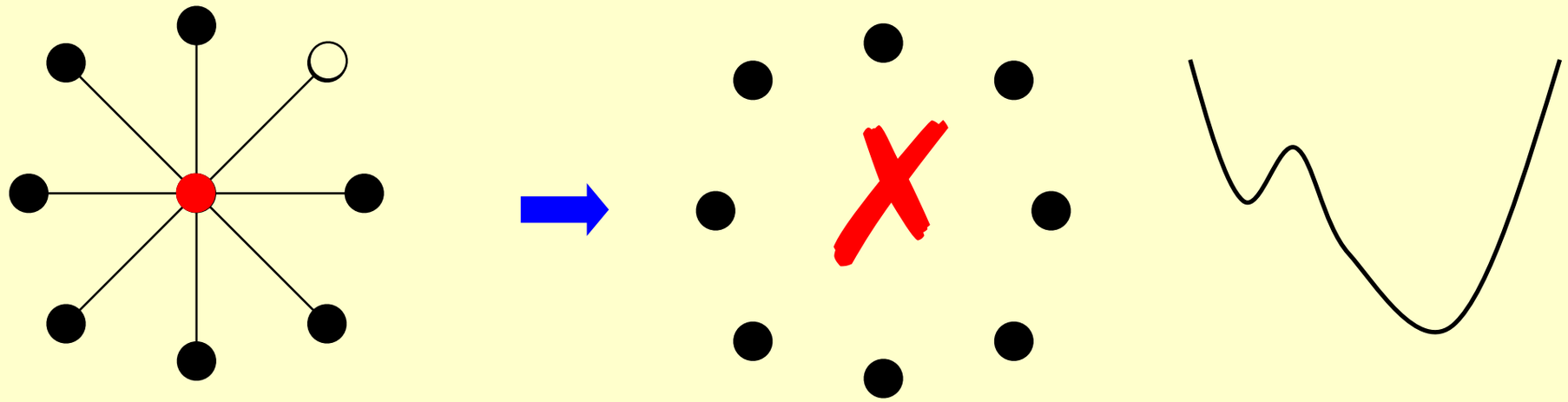

**Case 1:**



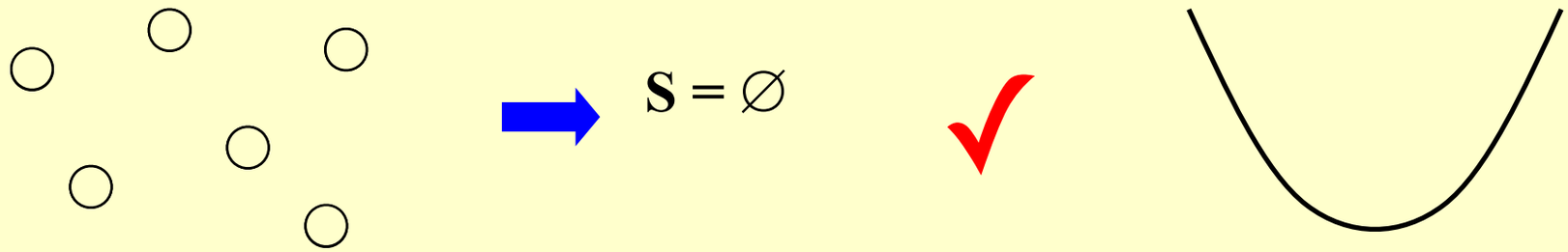
**Case 0:**



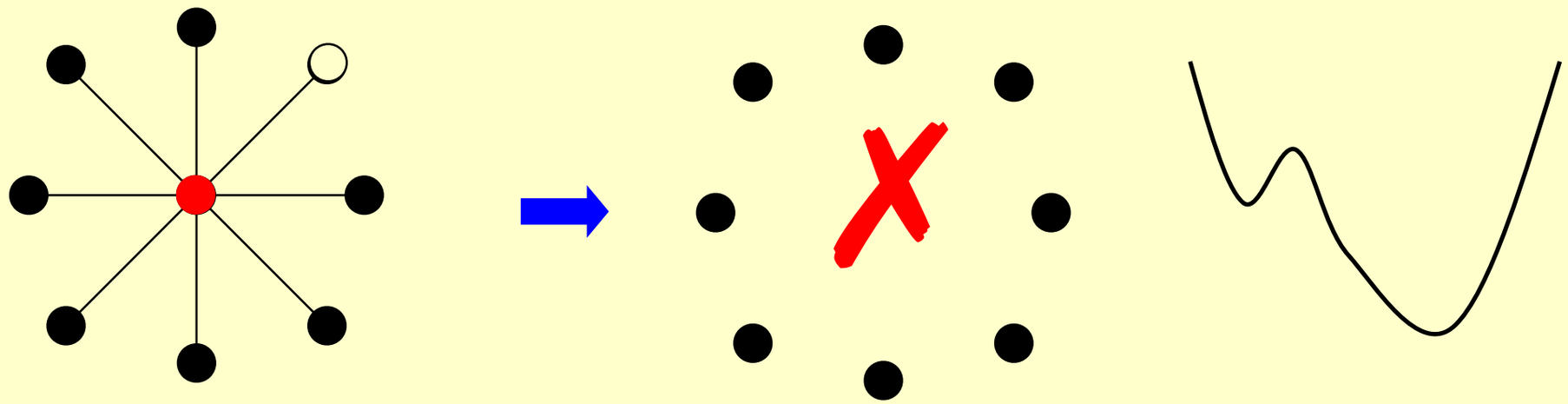

**Case 1:**



**Case 0:**

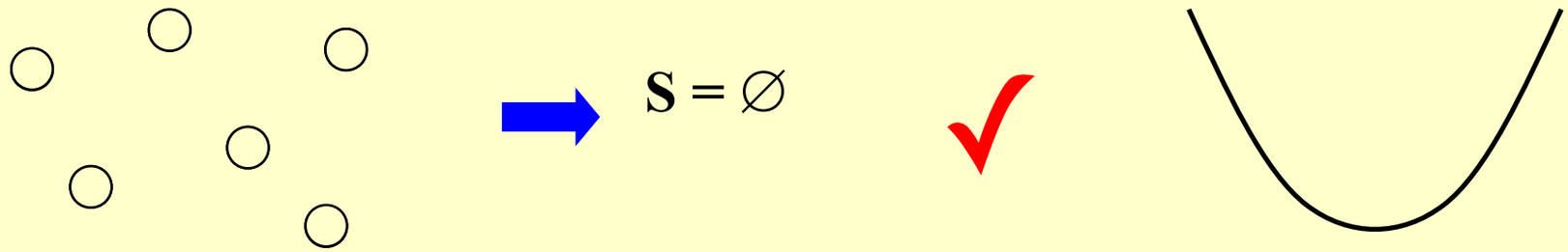



**Case 1:**

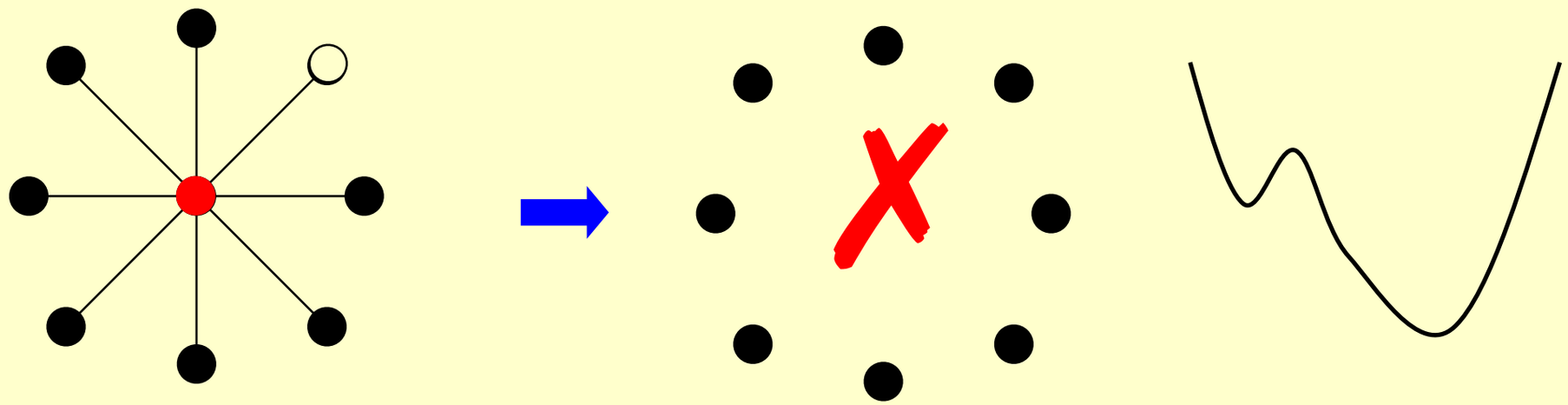


**Case 2:**

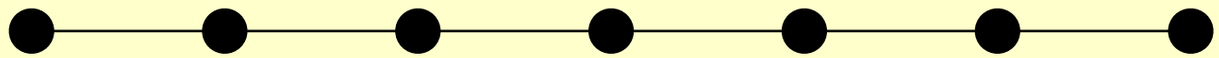
**Case 0:**



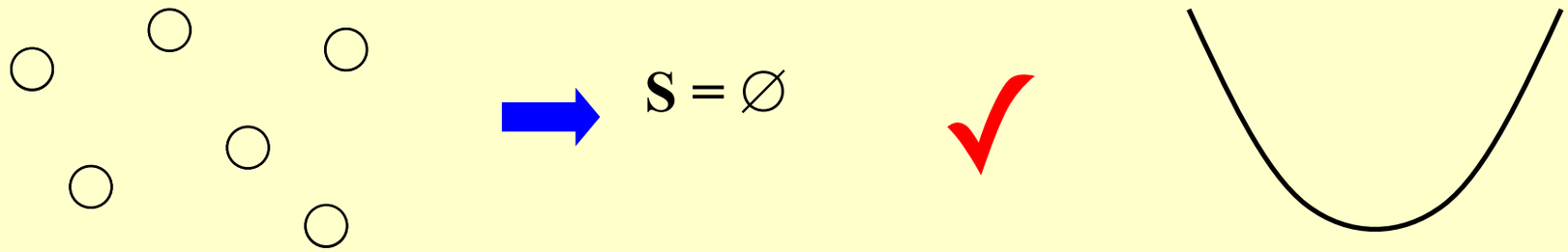

**Case 1:**



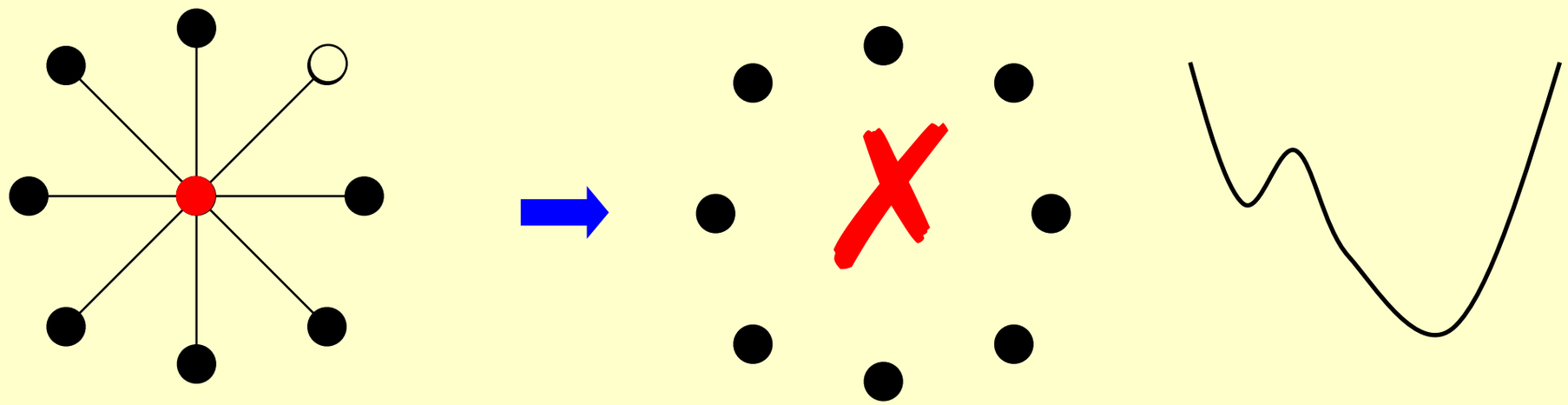
**Case 2:**



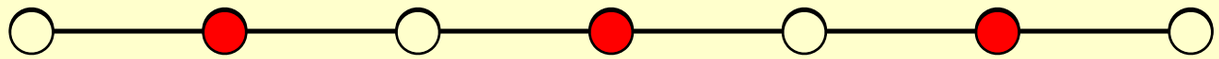
**Case 0:**



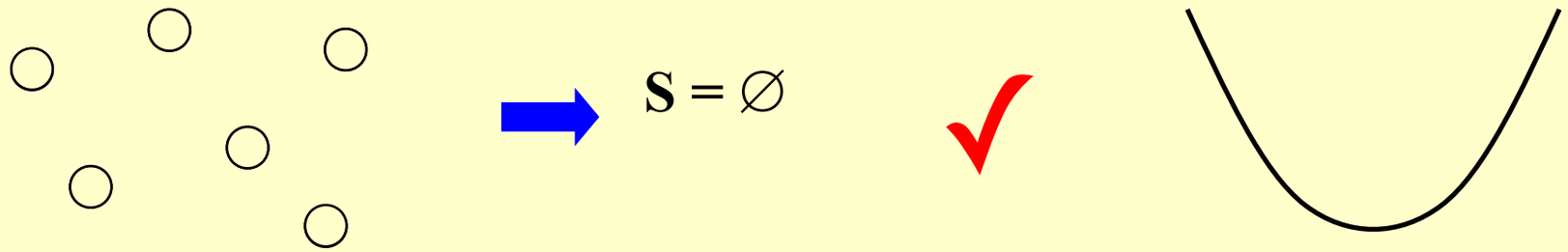

**Case 1:**



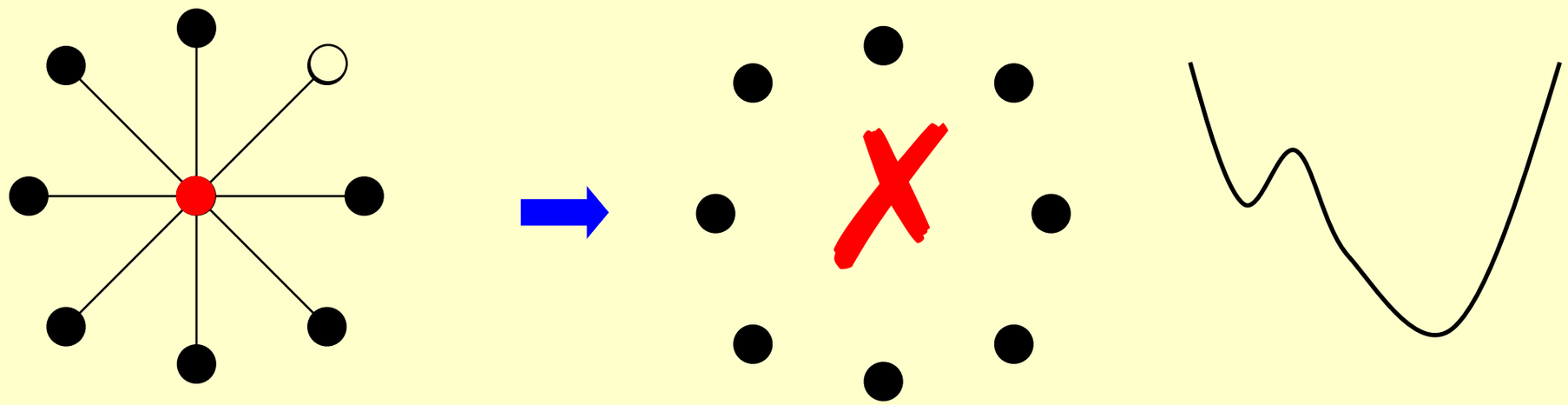
**Case 2:**



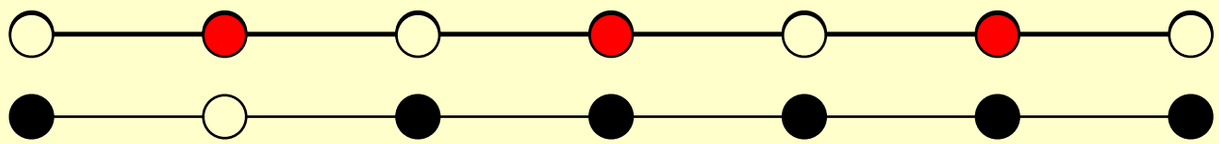
**Case 0:**



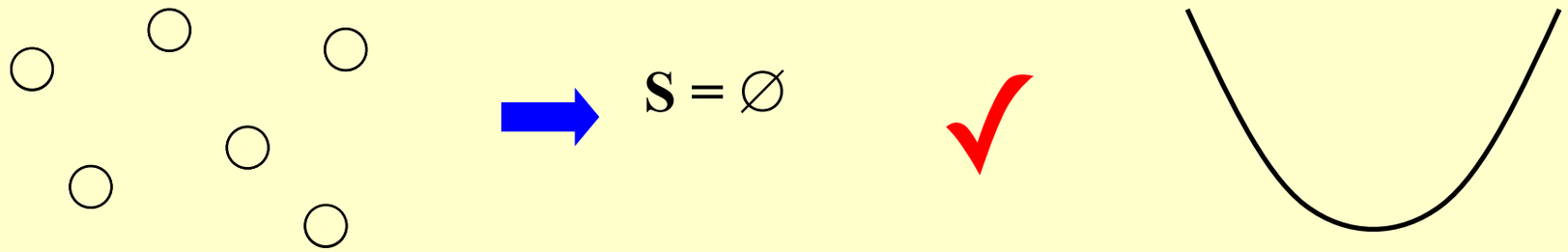

**Case 1:**



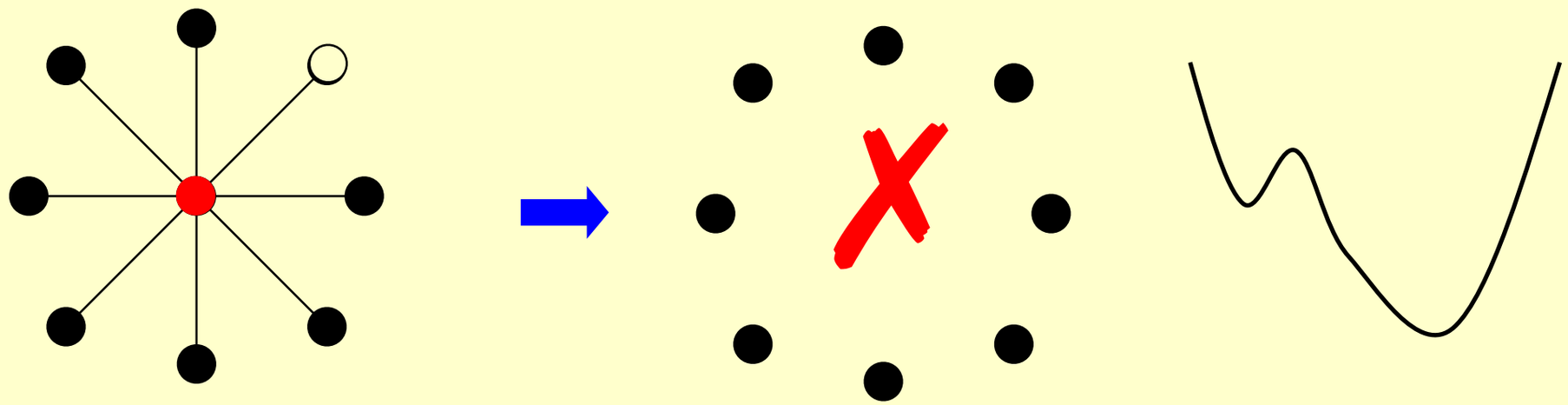
**Case 2:**



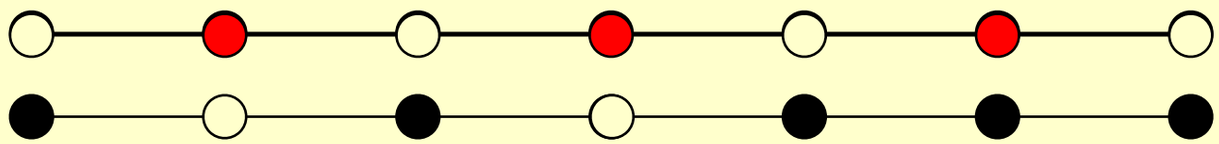
**Case 0:**



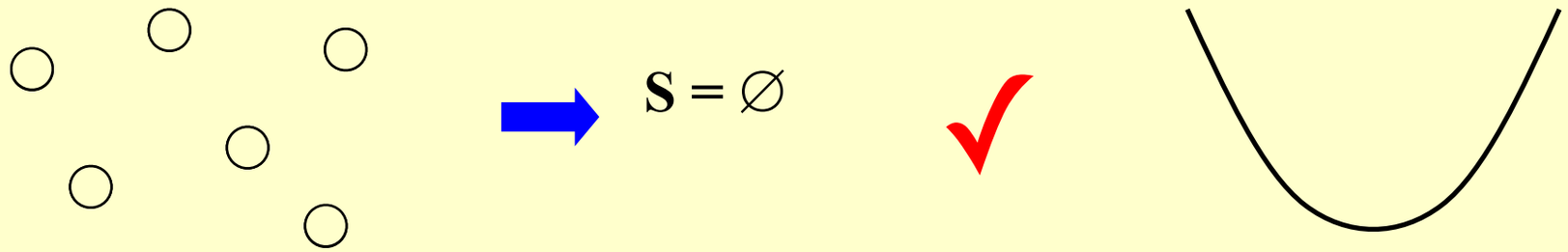

**Case 1:**



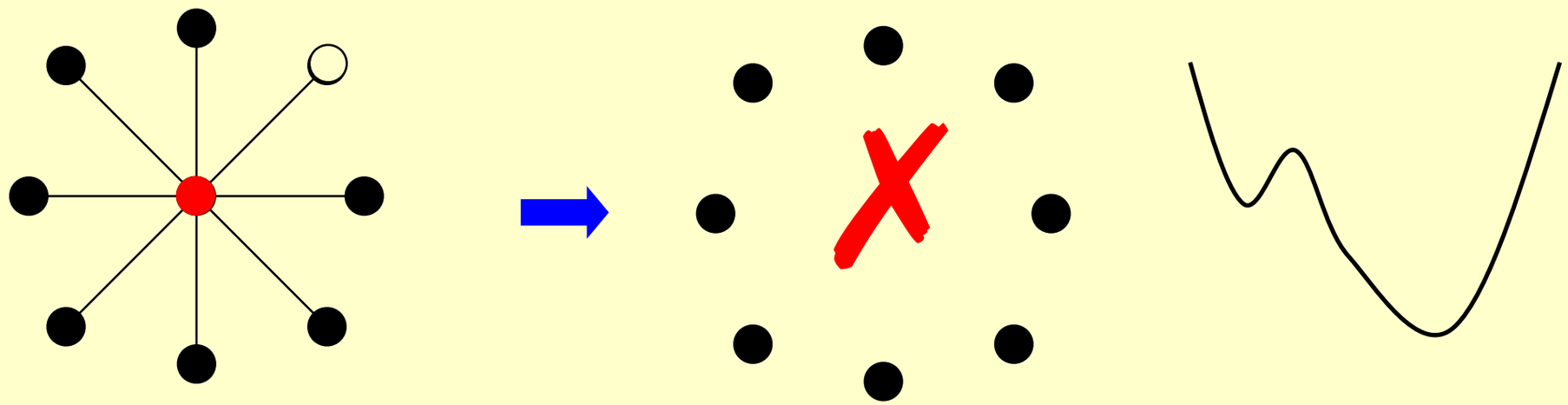
**Case 2:**



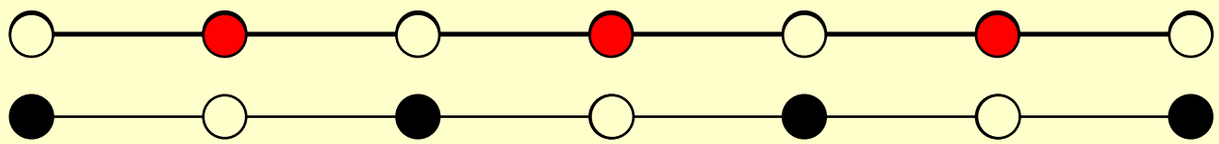
**Case 0:**



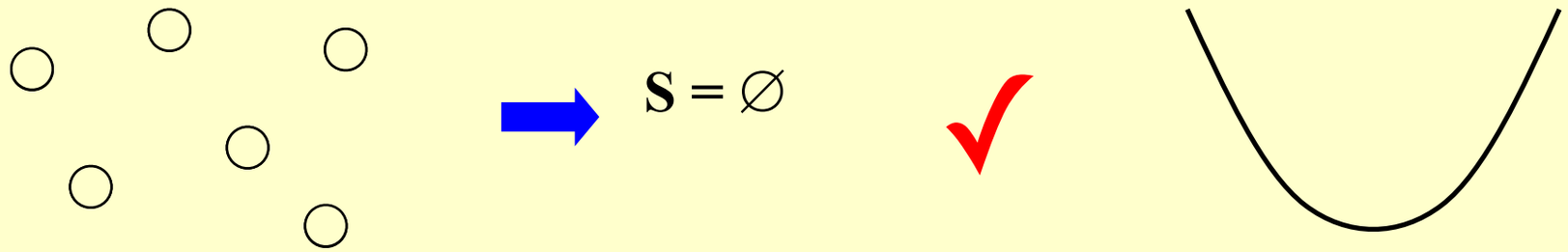

**Case 1:**



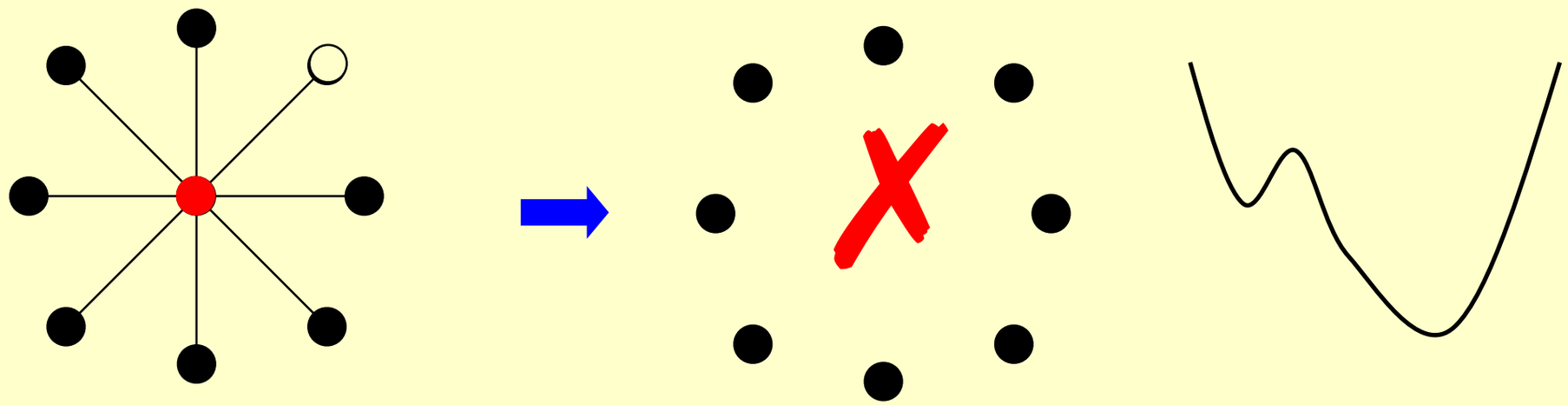
**Case 2:**



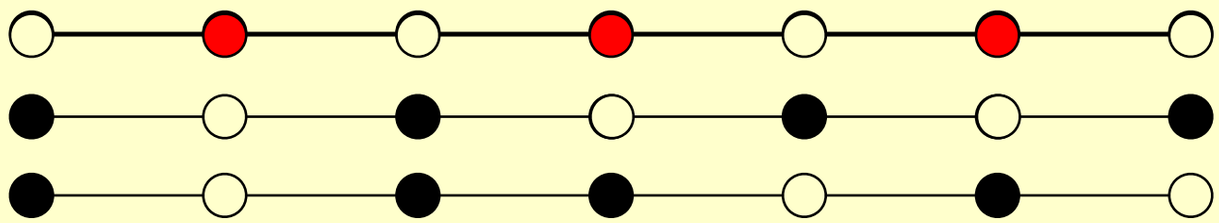
**Case 0:**



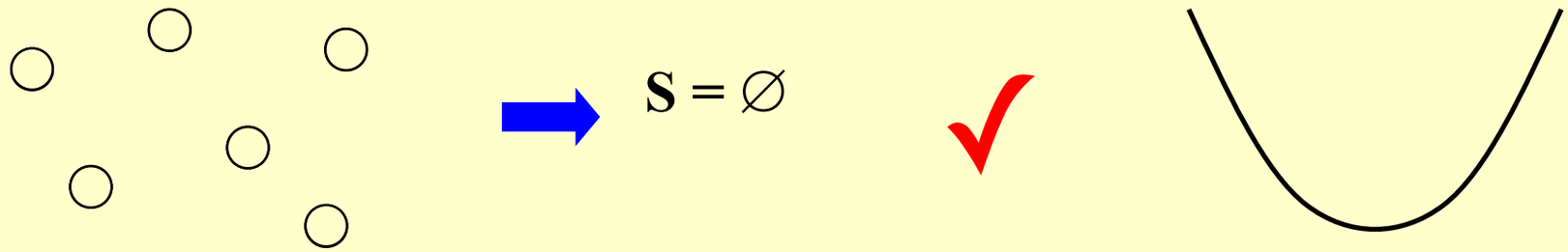

**Case 1:**



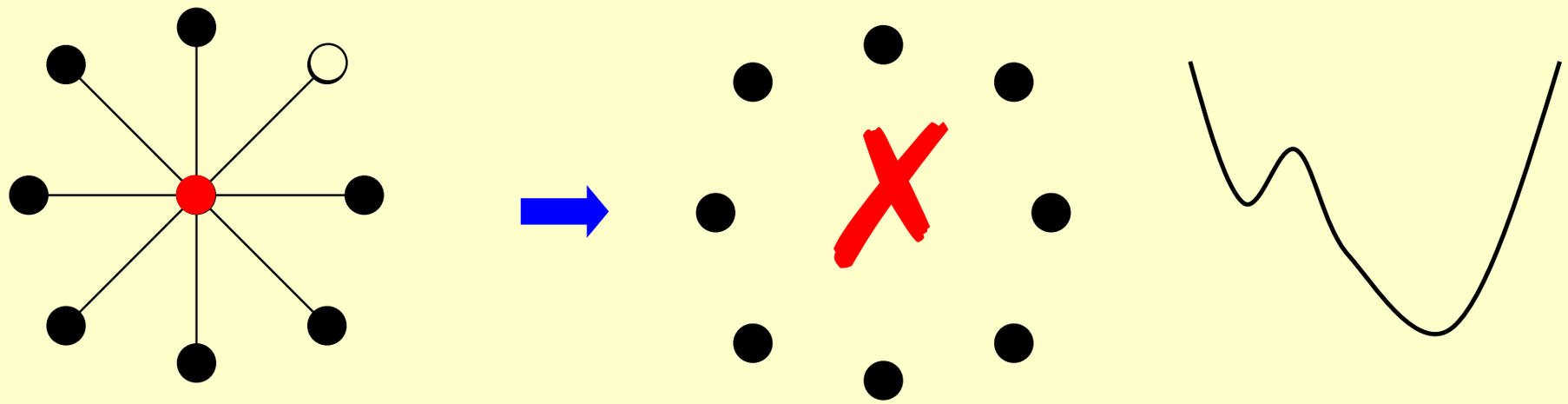
**Case 2:**



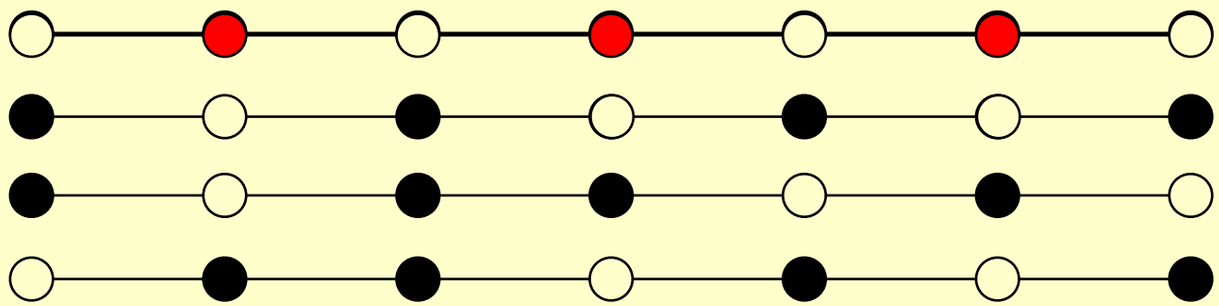
**Case 0:**



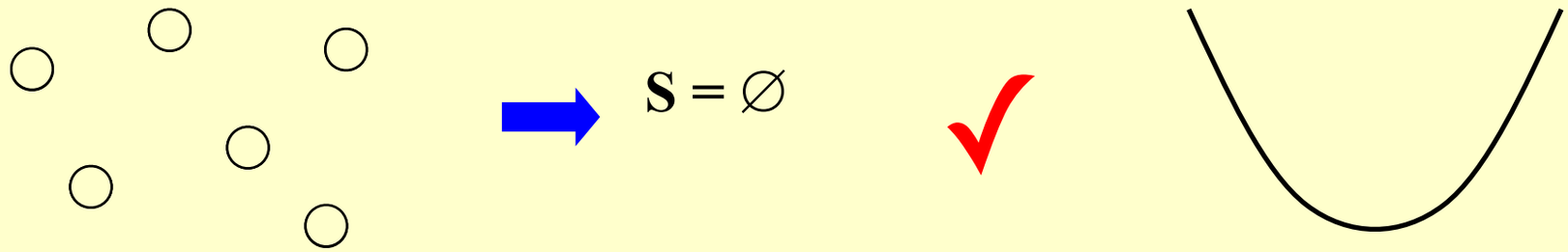

**Case 1:**



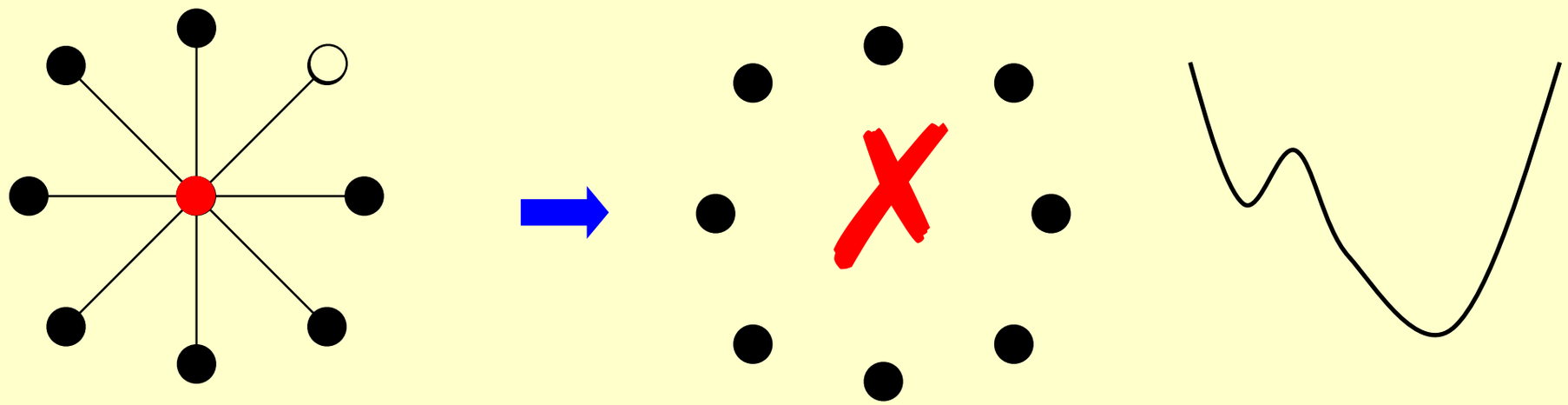
**Case 2:**



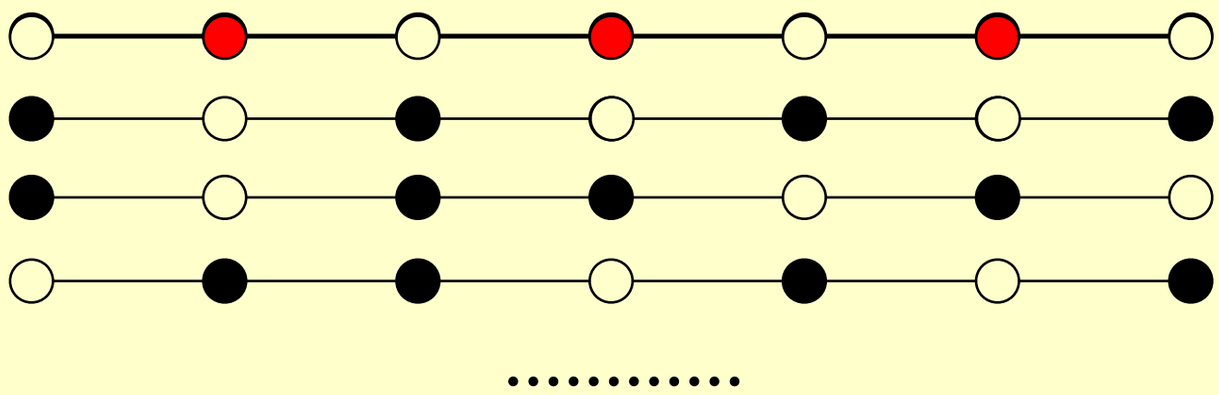
**Case 0:**



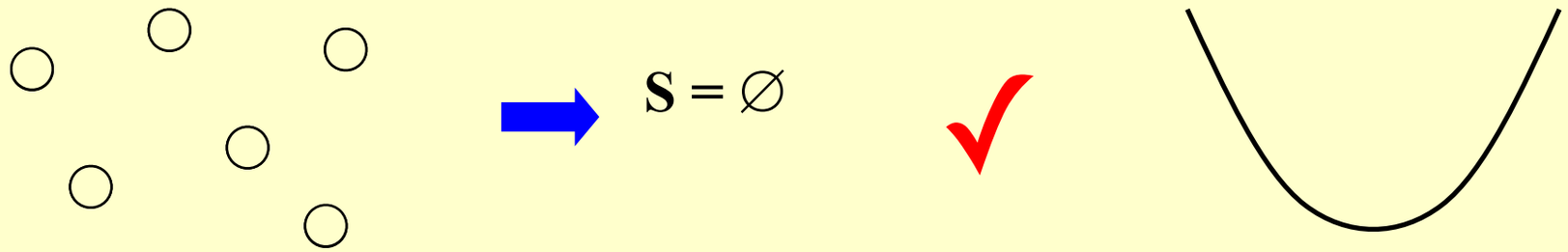

**Case 1:**



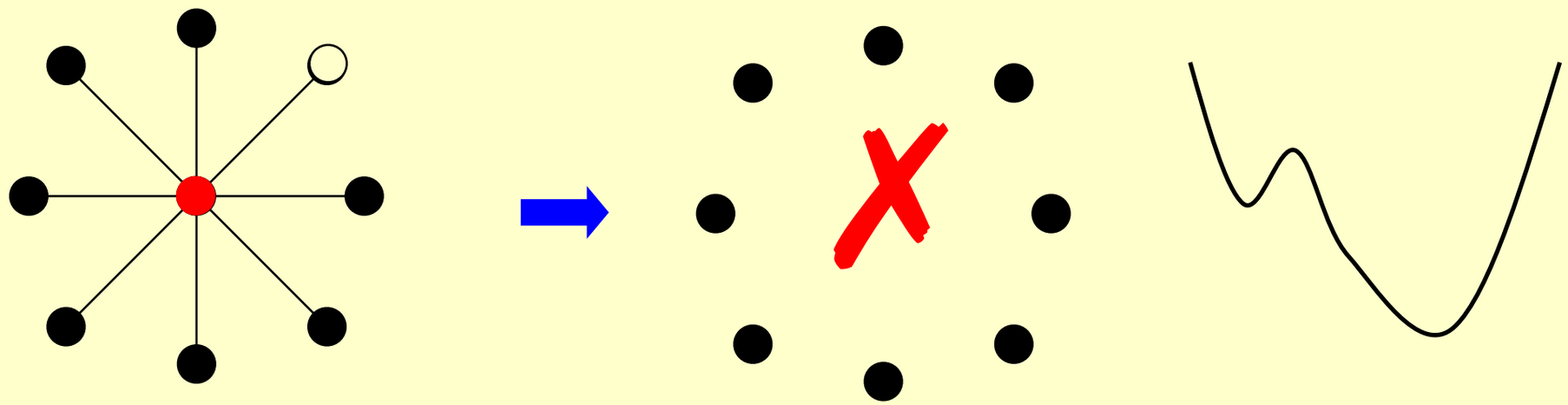
**Case 2:**



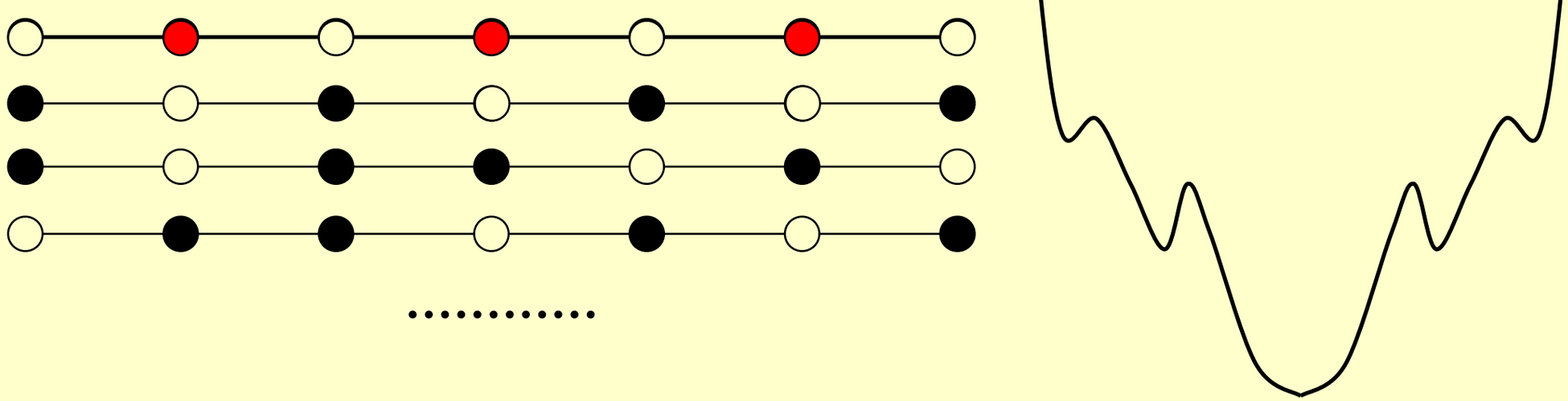
**Case 0:**



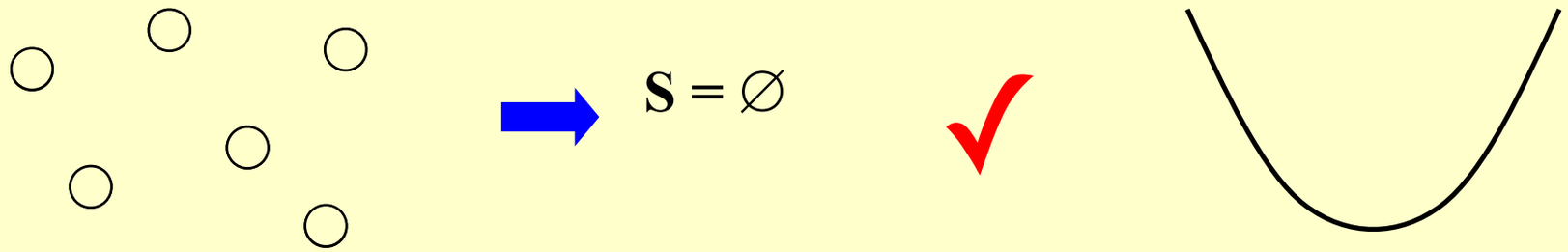

**Case 1:**



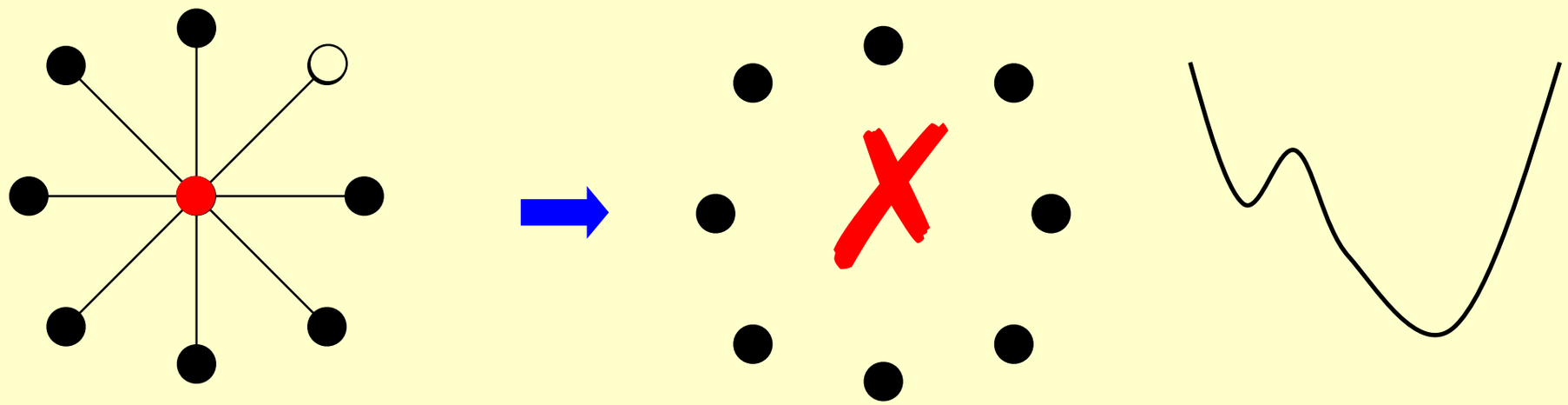
**Case 2:**



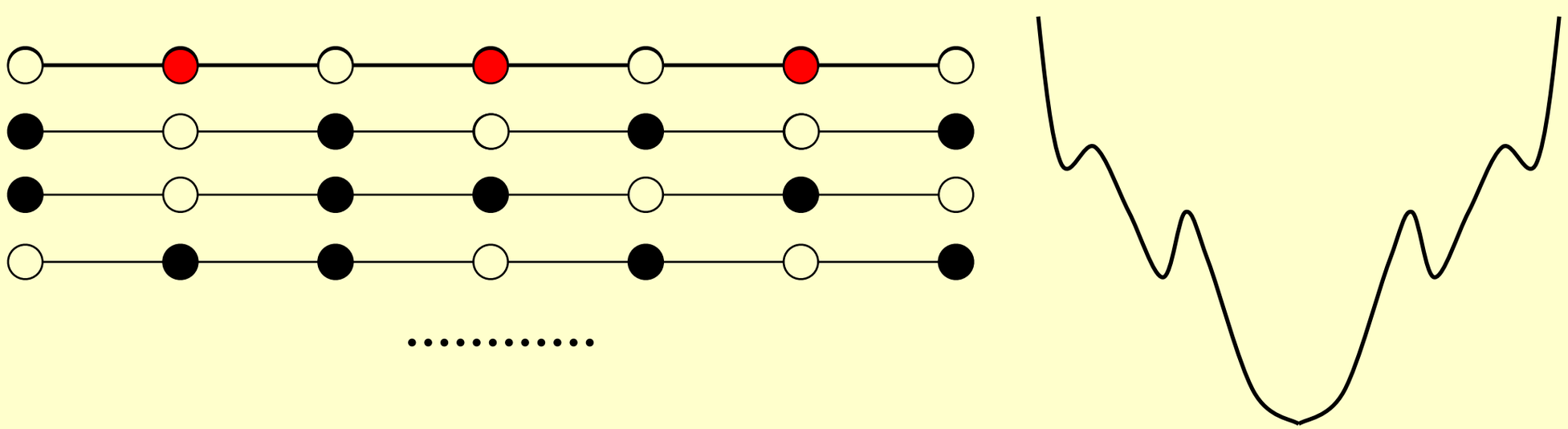
**Case 0:**




**Case 1:**



**Case 2:**



**Discussion 17:**

**Can you give another case in which gradient descent doesn't work?**

# Try to improve …

# Try to improve …

## ☞ The Metropolis Algorithm

```
SolutionType Metropolis()
{   Define constants k and T;

    Start from a feasible solution S ∈ FS ;

    MinCost = cost(S);
    while (1) {
        S' = Randomly chosen from N(S);
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else {
            With a probability e^(-Δcost/(kT)), let S = S';
            else  break;
        }
    }
    return S;
}
```

# Try to improve …

## ☞ The Metropolis Algorithm

```
SolutionType Metropolis()
{   Define constants k and T;

    Start from a feasible solution S ∈ FS ;

    MinCost = cost(S);
    while (1) {
        S' = Randomly chosen from N(S);
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else {
            With a probability e^(-Δcost/(kT)), let S = S';
            else  break;
        }
    }
    return S;
}
```

Adding is allowed

# ☞ Simulated Annealing

# ☞ Simulated Annealing

## ☞ Simulated Annealing

**The material is cooled *very gradually* from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures.**

## ☞ Simulated Annealing

**The material is cooled *very gradually* from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures.**

**Cooling schedule:** $T = \{ T_1 , T_2 , \ldots \}$

## ☞ Simulated Annealing

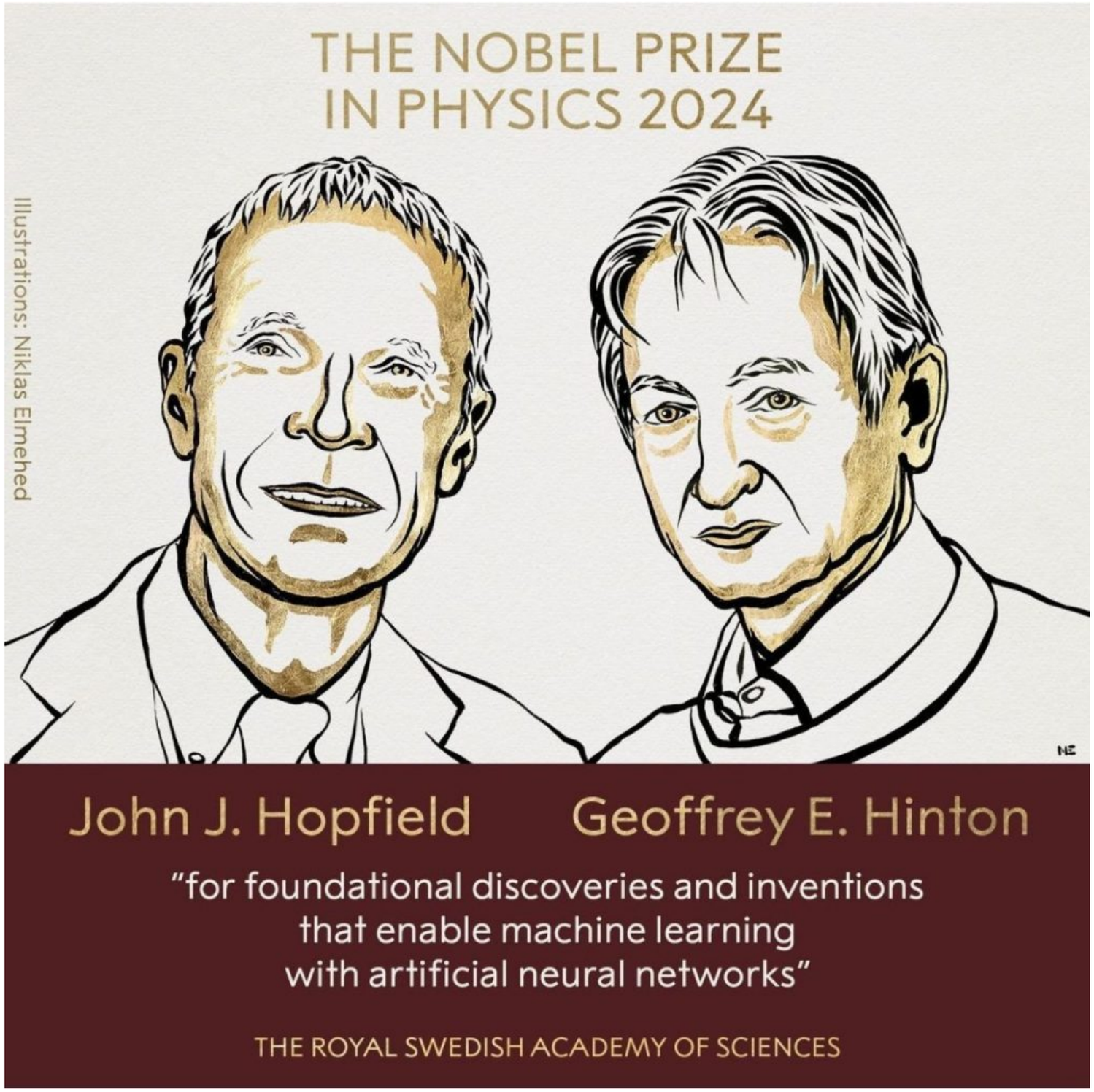
**The material is cooled *very gradually* from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures.**

**Cooling schedule:** $T = \{ T_1 , T_2 , \ldots \}$

# Outline: Local Search

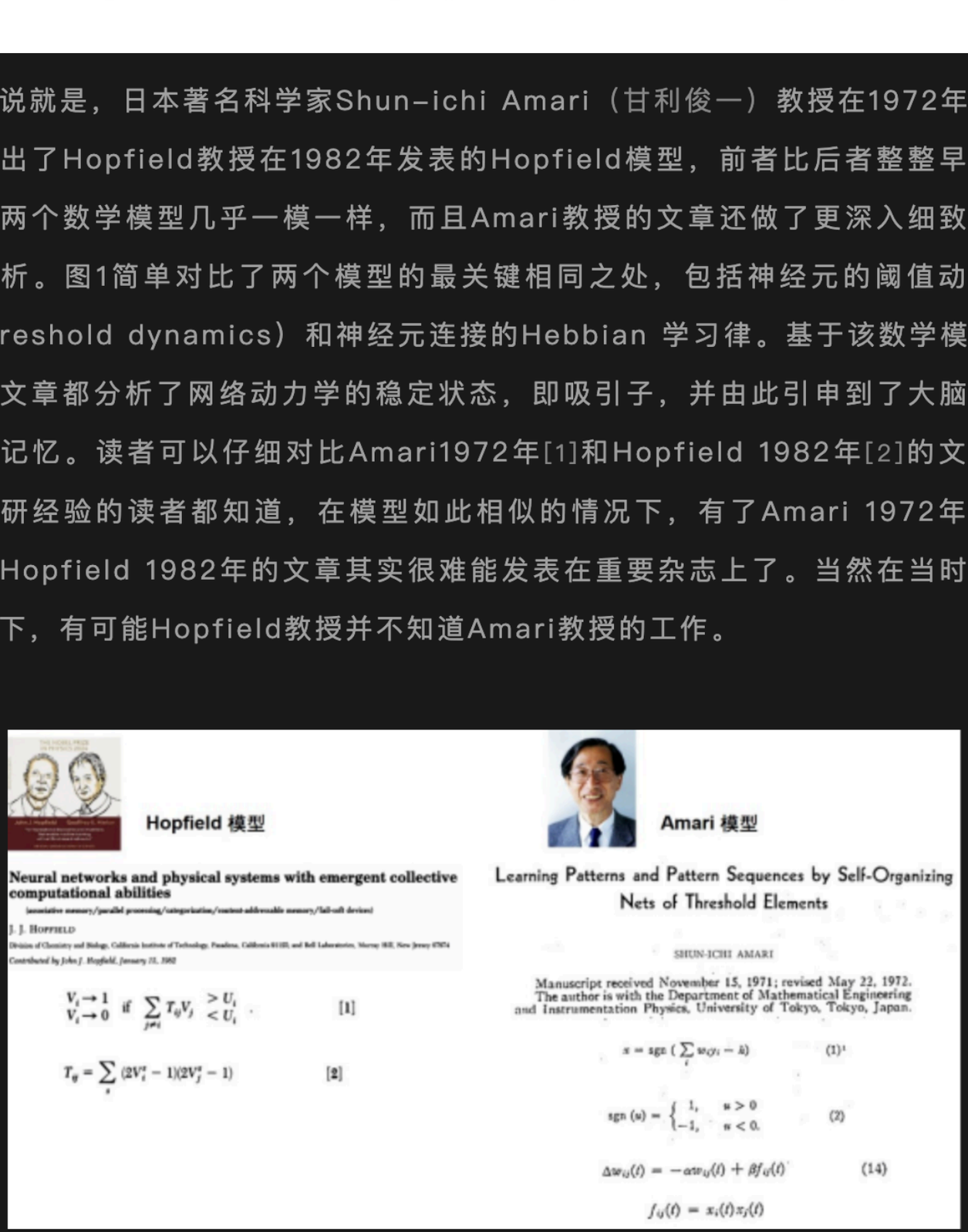
- Vertex cover
- Hopfield neural networks
- Max-cut
- Take-home messages

# Hopfield Neural Networks



# Hopfield Neural Networks

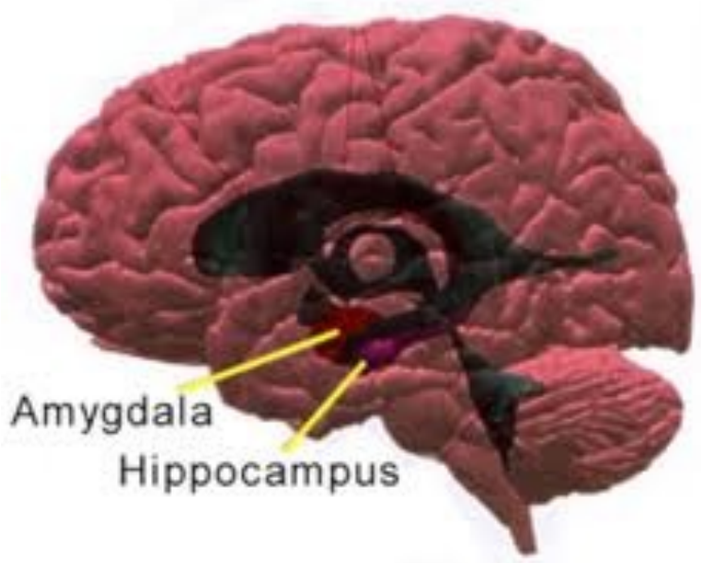
简单说就是，日本著名科学家Shun-ichi Amari（甘利俊一）教授在1972年就已经提出了Hopfield教授在1982年发表的Hopfield模型，前者比后者整整早了十年。两个数学模型几乎一模一样，而且Amari教授的文章还做了更深入细致的数学分析。图1简单对比了两个模型的最关键相同之处，包括神经元的阈值动力学（threshold dynamics）和神经元连接的Hebbian 学习律。基于该数学模型，两篇文章都分析了网络动力学的稳定状态，即吸引子，并由此引申到了大脑的联想式记忆。读者可以仔细对比Amari1972年[1]和Hopfield 1982年[2]的文章。有科研经验的读者都知道，在模型如此相似的情况下，有了Amari 1972年的文章，Hopfield 1982年的文章其实很难能发表在重要杂志上了。当然在当时资讯条件下，有可能Hopfield教授并不知道Amari教授的工作。

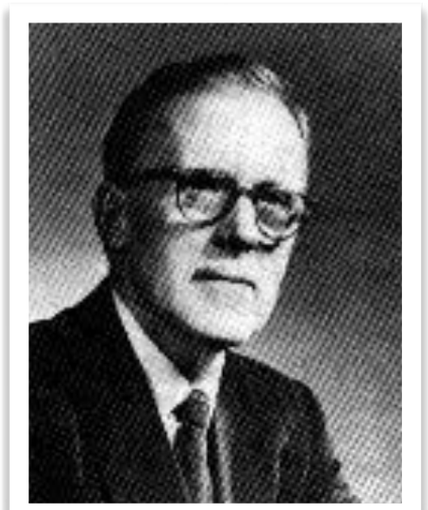


# Hopfield Neural Networks

## Human learning

- Learning is to associate two events with each other.
- In the Hebbian type of learning, both presynaptic and postsynaptic neurons are involved.
- The main brain organ for learning/explicit memory is the hippocampus (of the limbic system) using Hebbian type.

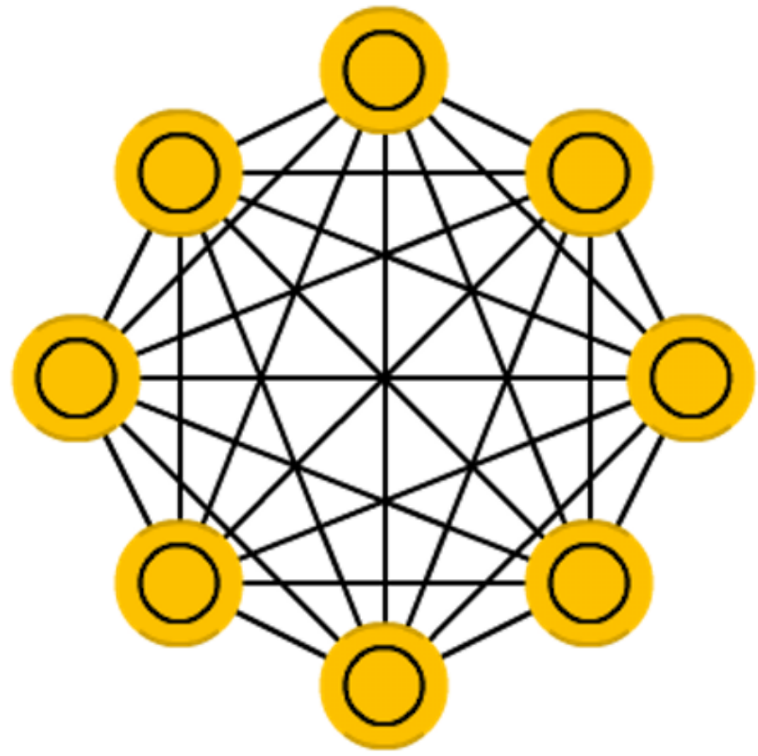


**Donald O. Hebb**

# Hopfield Neural Networks

- Fully-connect undirected graphs.
- each node represents binary states -1 or +1.
- each edge has a weight representing the strength of association between two end nodes.



# Hopfield Neural Networks

| |
|---|
| moscow------russia |
| lima----------peru |
| london-----england |
| tokyo--------japan |
| edinburgh-scotland |
| ottawa------canada |
| oslo--------norway |
| stockholm---sweden |
| paris-------france |

desired associative memory

moscow---:::::::::  $\Longrightarrow$  moscow------russia

::::::::::--canada  $\Longrightarrow$  ottawa------canada

functionality 1: pattern completion

otowa-------canada  $\Longrightarrow$  ottawa------canada

egindurrh-sxotland  $\Longrightarrow$  edinburgh-scotland

functionality 2: error correction

# Hopfield Neural Networks

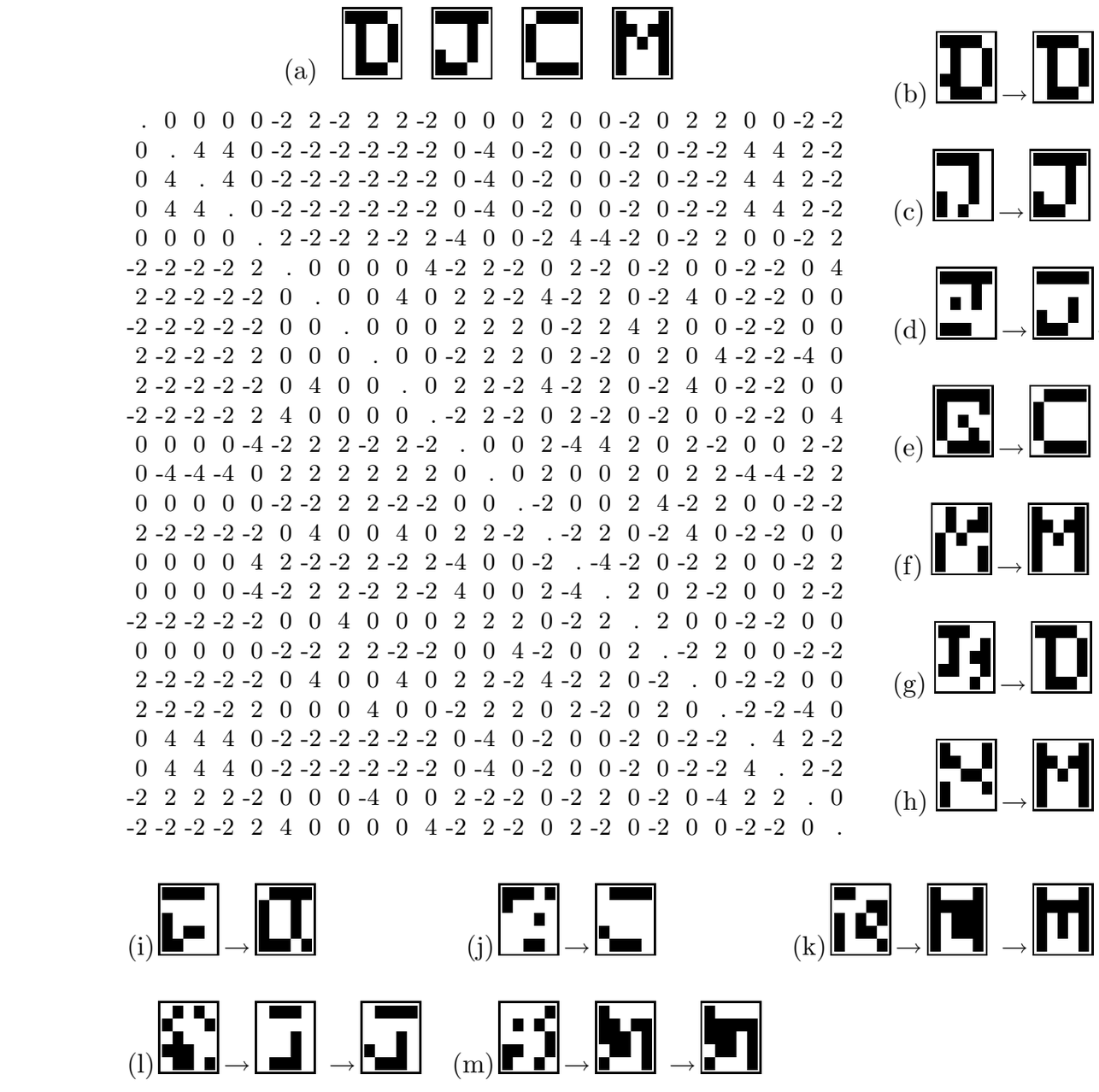



Figure 42.3. Binary Hopfield network storing four memories. (a) The four memories, and the weight matrix. (b–h) Initial states that differ by one, two, three, four, or even five bits from a desired memory are restored to that memory in one or two iterations. (i–m) Some initial conditions that are far from the memories lead to stable states other than the four memories; in (i), the stable state looks like a mixture of two memories, 'D' and 'J'; stable state (j) is like a mixture of 'J' and 'C'; in (k), we find a corrupted version of the 'M' memory (two bits distant); in (l) a corrupted version of 'J' (four bits distant) and in (m), a state which looks spurious until we recognize that it is the inverse of the stable state (l).

More details on Hopfield nets see Mackay book Chap. 42.

# Attention Is All You Need

**Ashish Vaswani*** Google Brain avaswani@google.com

**Noam Shazeer*** Google Brain noam@google.com

**Niki Parmar*** Google Research nikip@google.com

**Jakob Uszkoreit*** Google Research usz@google.com

**Llion Jones*** Google Research llion@google.com

**Aidan N. Gomez*** † University of Toronto aidan@cs.toronto.edu

**Łukasz Kaiser*** Google Brain lukaszkaiser@google.com

**Illia Polosukhin*** ‡ illia.polosukhin@gmail.com

# HOPFIELD NETWORKS IS ALL YOU NEED

**Hubert Ramsauer*** **Bernhard Schäfl*** **Johannes Lehner*** **Philipp Seidl***
**Michael Widrich*** **Thomas Adler*** **Lukas Gruber*** **Markus Holzleitner***
**Milena Pavlović**[‡,§] **Geir Kjetil Sandve**[§] **Victor Greiff**[‡] **David Kreil**[†]
**Michael Kopp**[†] **Günter Klambauer*** **Johannes Brandstetter*** **Sepp Hochreiter***[,†]

*ELLIS Unit Linz, LIT AI Lab, Institute for Machine Learning,
Johannes Kepler University Linz, Austria
†Institute of Advanced Research in Artificial Intelligence (IARAI)
‡Department of Immunology, University of Oslo, Norway
§Department of Informatics, University of Oslo, Norway

# Hopfield neural networks

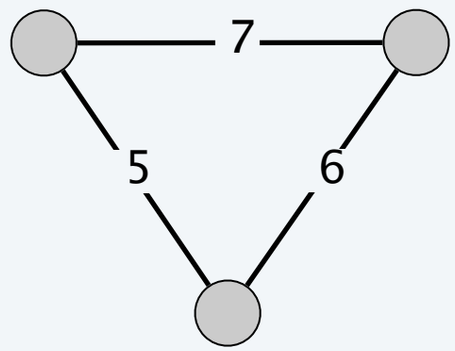
Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

Input: Graph $G = (V, E)$ with integer (positive or negative) edge weights $w$.

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then $u$ and $v$ want to have the same state; if $w_{uv} > 0$ then $u$ and $v$ want different states.

Note. In general, no configuration respects all constraints.



# Hopfield neural networks

Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

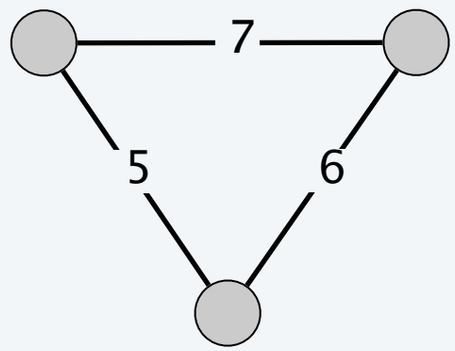
Input: Graph $G = (V, E)$ with integer (positive or negative) edge weights $w$.

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then $u$ and $v$ want to have the same state; if $w_{uv} > 0$ then $u$ and $v$ want different states.

Note. In general, no configuration respects all constraints.

In actual Hopfield nets, the situation is the opposite: the end nodes have the same state when the edge weight is positive.



# Hopfield neural networks

Def. With respect to a configuration $S$, edge $e = (u, v)$ is good if $w_e \times s_u \times s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, then $s_u \neq s_v$.

# Hopfield neural networks

Def. With respect to a configuration $S$, edge $e = (u, v)$ is good if $w_e \times s_u \times s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, then $s_u \neq s_v$.

Def. With respect to a configuration $S$, a node $u$ is satisfied if the weight of incident good edges ≥ weight of incident bad edges.

$$\sum_{v:\ e=(u,v)\in E} w_e\, s_u\, s_v \ \leq\ 0$$

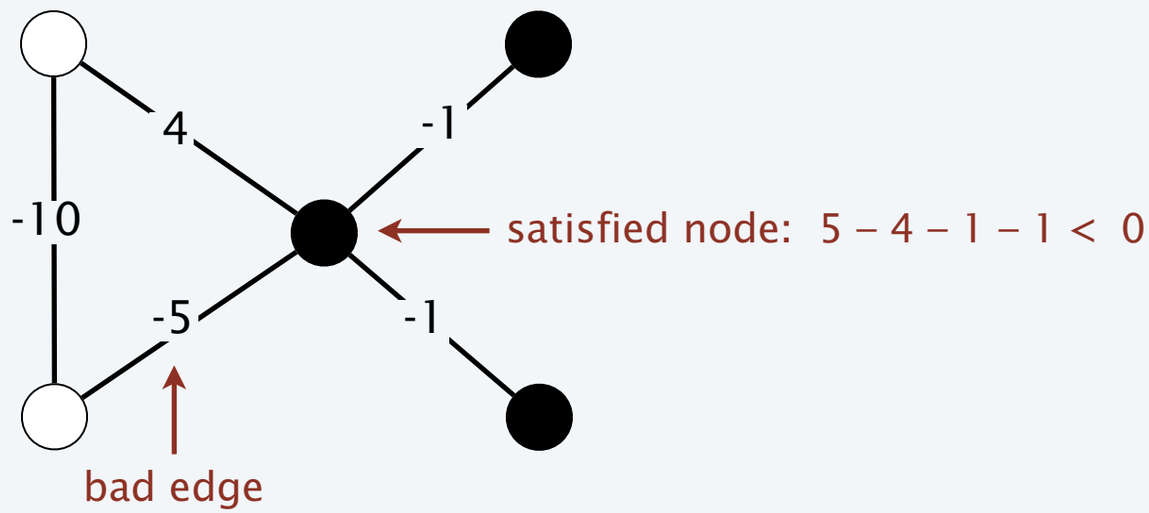
# Hopfield neural networks

Def. With respect to a configuration $S$, edge $e = (u, v)$ is good if $w_e \times s_u \times s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, then $s_u \neq s_v$.

Def. With respect to a configuration $S$, a node $u$ is satisfied if the weight of incident good edges ≥ weight of incident bad edges.

$$\sum_{v:\ e=(u,v)\in E} w_e\, s_u\, s_v \ \leq\ 0$$

Def. A configuration is stable if all nodes are satisfied.




Goal. Find a stable configuration, if such a configuration exists.

# Hopfield neural networks

Goal. Find a stable configuration, if such a configuration exists.

State-flipping algorithm. Repeated flip state of an unsatisfied node.

*HOPFIELD-FLIP* $(G, w)$

$S \leftarrow$ *arbitrary configuration*.

*WHILE* *(current configuration is not stable)*

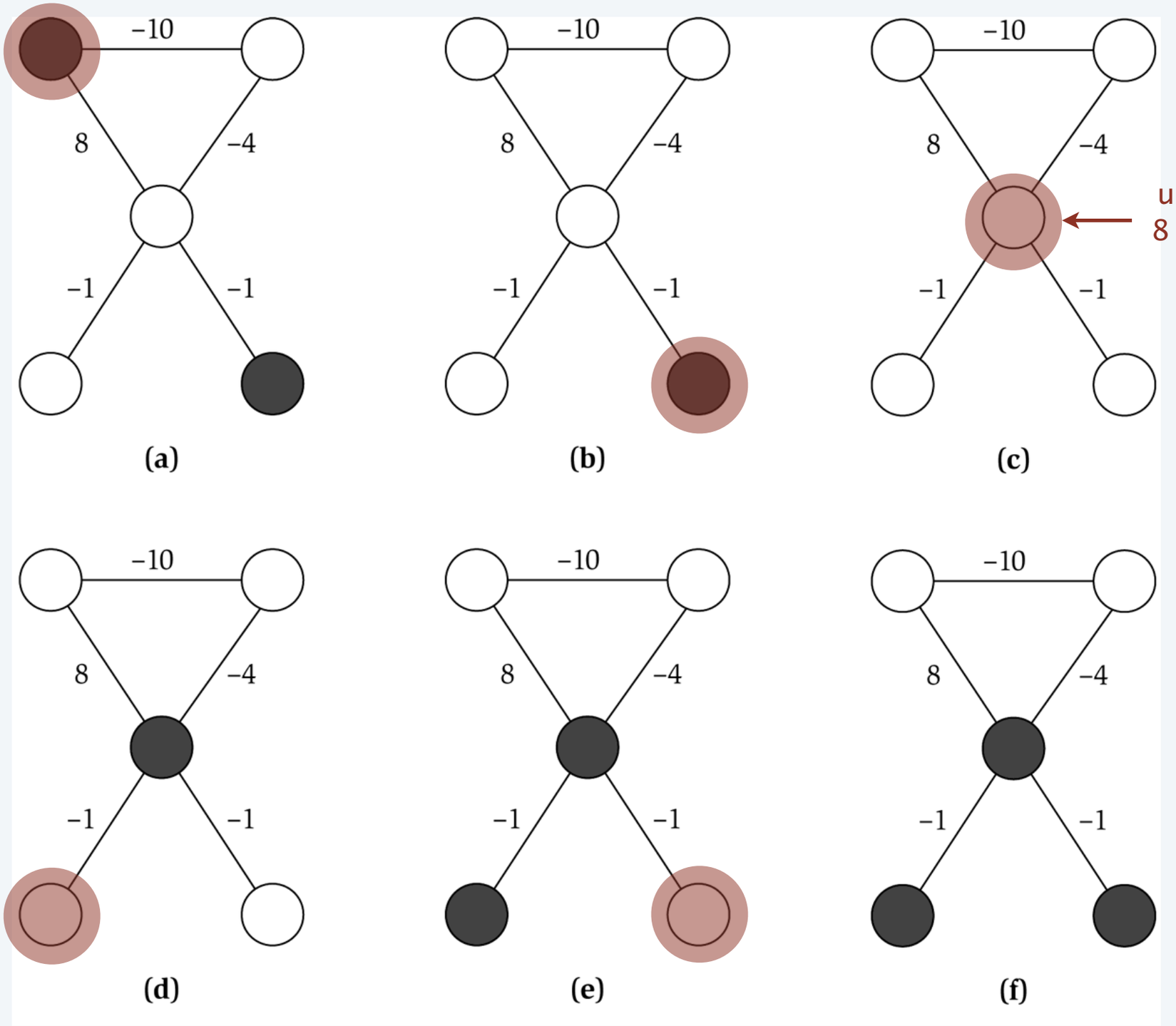
  $u \leftarrow$ *unsatisfied node*.

  $s_u \leftarrow -s_u$.

*RETURN* $S$.

# State-flipping algorithm example

unsatisfied node
10 − 8 > 0




unsatisfied node
8 − 4 − 1 − 1 > 0

stable

# State-flipping algorithm: proof of correctness

Theorem. The state-flipping algorithm terminates with a stable configuration after at most $W = \Sigma_e |w_e|$ iterations.

Pf. Consider measure of progress $\Phi(S) = \Sigma_{e \text{ good}} |w_e|$.

- Clearly $0 \leq \Phi(S) \leq W$.
- We show $\Phi(S)$ increases by at least $1$ after each flip.

# State-flipping algorithm: proof of correctness

Theorem. The state-flipping algorithm terminates with a stable configuration after at most $W = \Sigma_e |w_e|$ iterations.

Pf. Consider measure of progress $\Phi(S) = \Sigma_{e \text{ good}} |w_e|$.

- Clearly $0 \leq \Phi(S) \leq W$.
- We show $\Phi(S)$ increases by at least $1$ after each flip.

  When $u$ flips state:
  - all good edges incident to $u$ become bad
  - all bad edges incident to $u$ become good
  - all other edges remain the same

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is good}}} |w_e| \geq \Phi(S) + 1$$

u is unsatisfied

# State-flipping algorithm: proof of correctness

Theorem. The state-flipping algorithm terminates with a stable configuration after at most $W = \Sigma_e |w_e|$ iterations.

Pf. Consider measure of progress $\Phi(S) = \Sigma_{e \text{ good}} |w_e|$.

- Clearly $0 \leq \Phi(S) \leq W$.
- We show $\Phi(S)$ increases by at least $1$ after each flip.

  When $u$ flips state:
  - all good edges incident to $u$ become bad
  - all bad edges incident to $u$ become good
  - all other edges remain the same

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is good}}} |w_e| \geq \Phi(S) + 1$$

u is unsatisfied

Showing one-step improvement is usually an essential step for proving the correctness and the convergence rate of optimization.

# Complexity of Hopfield neural network

Hopfield network search problem. Given a weighted graph, find a stable configuration if one exists.

Hopfield network decision problem. Given a weighted graph, does there exist a stable configuration?

Remark. The decision problem is trivially solvable (always yes), but there is no known poly-time algorithm for the search problem.

polynomial in n and log W

# Outline: Local Search

- Vertex cover
- Hopfield neural networks
- Max-cut
- Take-home messages

## 〖Example〗 The Maximum Cut Problem.

## 〖Example〗 The Maximum Cut Problem.

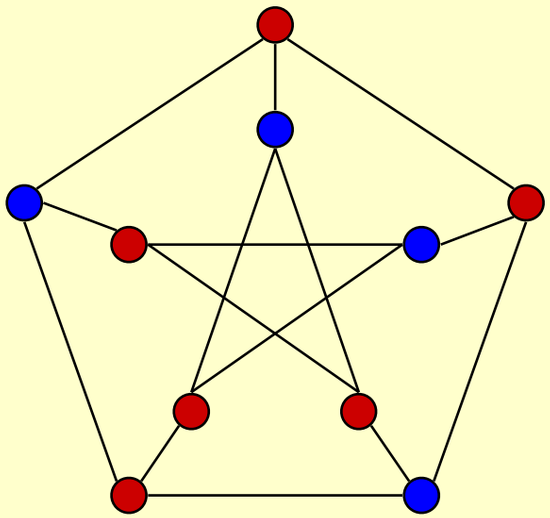
❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights $w_e$, find a node partition $(A, B)$ such that the total weight of edges crossing the cut is maximized.

$$w(A,B) := \sum_{u \in A,\, v \in B} w_{uv}$$

## 〖Example〗 The Maximum Cut Problem.

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights $w_e$, find a node partition $(A, B)$ such that the total weight of edges crossing the cut is maximized.

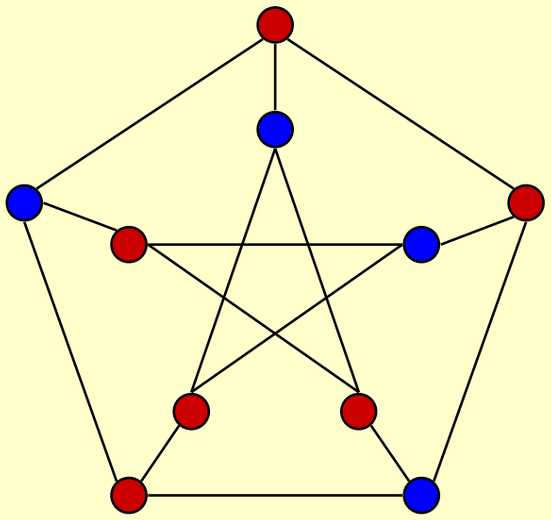
$$w(A,B) := \sum_{u \in A,\, v \in B} w_{uv}$$



## 〖Example〗 The Maximum Cut Problem.

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights $w_e$, find a node partition $(A, B)$ such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u \in A,\, v \in B} w_{uv}$$

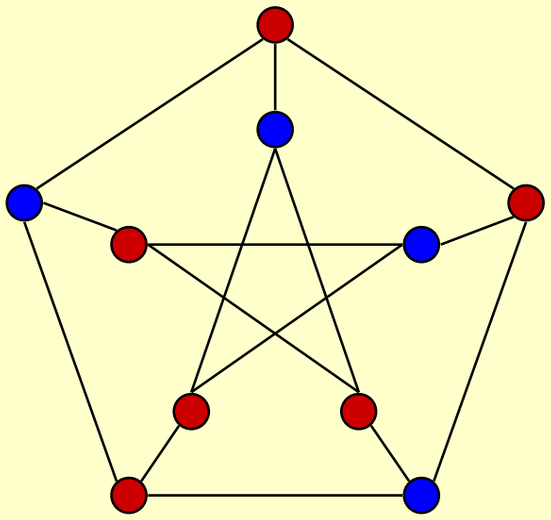


- **Toy application**
  - $n$ activities, $m$ people.
  - Each person wants to participate in two of the activities.
  - Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.

## 〖Example〗 The Maximum Cut Problem.

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights $w_e$, find a node partition $(A, B)$ such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u \in A,\, v \in B} w_{uv}$$



- **Toy application**
  - $n$ activities, $m$ people.
  - Each person wants to participate in two of the activities.
  - Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.
- **Real applications** Circuit layout, statistical physics

# Related to Local Search

## Related to Local Search

☞ **Problem: To *maximize* $w(A, B)$.**

## Related to Local Search

- ☞ **Problem: To *maximize* $w(A, B)$.**
- ☞ **Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$**

## Related to Local Search

- ☞ **Problem: To *maximize* $w(A, B)$.**
- ☞ **Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$**
- ☞ **S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.**

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximize* $w(A, B)$.
- Feasible solution set $\mathcal{FS}$ : any partition ($A$, $B$)
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximize* $w(A, B)$.
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

☝ A special case of Hopfield Neural Network – with $w_e$ all being positive!

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximiz* $\Phi(S) = \Sigma_{e \text{ is } \textit{good}} |w_e|$
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

A special case of Hopfield Neural Network – with $w_e$ all being positive!

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximiz* $\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

A special case of Hopfield Neural Network – with $w_e$ all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximiz* $\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

A special case of Hopfield Neural Network – with $w_e$ all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

*May NOT in polynomial time*

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximiz* $\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

A special case of Hopfield Neural Network – with $w_e$ all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

*May NOT in polynomial time*

- How good is this local optimum?

## Related to Local Search

*Single-flip neighborhood*

- Problem: To *maximiz* $\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$
- Feasible solution set $\mathcal{FS}$ : any partition $(A, B)$
- S ~ S': S' can be obtained from S by moving one node from $A$ to $B$, or one from $B$ to $A$.

A special case of Hopfield Neural Network – with $w_e$ all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

*May NOT in polynomial time*

- How good is this local optimum?
- Try a better *local*?

- **How good is this local optimum?**

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2}\, w(A^*, B^*)$.**

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2}\, w(A^*, B^*)$.**

**Proof:**

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$\sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv}$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv}$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$\sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

- **How good is this local optimum?**

**Claim:** **Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.**

**Proof:** **Since $(A, B)$ is a local optimal partition, for any $u \in A$**

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

**Summing up for *all* $u \in A$**

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

- **How good is this local optimum?**

**Claim:** Let $(A, B)$ be a local optimal partition and let $(A^*, B^*)$ be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

**Proof:** Since $(A, B)$ is a local optimal partition, for any $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Summing up for *all* $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

∎

☞ **[Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.**

☞ **[Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.**

☞ **[Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.**

- **[Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.**

$$\min_{0 \le \theta \le \pi} \frac{\pi}{2} \frac{1-\cos\theta}{\theta}$$

- **[Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.**

- **[Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.**

$$\min_{0 \le \theta \le \pi} \frac{\pi}{2} \frac{1-\cos\theta}{\theta}$$

- **[Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.**

- **[Håstad 1997] Unless P = NP, no 17/16 approximation algorithm for MAX-CUT.**

- **[Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.**

$$\min_{0 \le \theta \le \pi} \frac{\pi}{2} \frac{1-\cos\theta}{\theta}$$

- **[Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.**

- **[Håstad 1997] Unless P = NP, no 17/16 approximation algorithm for MAX-CUT.**

1.0625

- **May NOT in polynomial time**

- **May NOT in polynomial time**

  ☝ **stop the algorithm when there are no "*big enough*" improvements.**

- **May NOT in polynomial time**

  ☝ **stop the algorithm when there are no "*big enough*" improvements.**

  **Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least**

$$\frac{2\varepsilon}{|V|} w(A,B)$$

- **May NOT in polynomial time**

  ☝ **stop the algorithm when there are no "*big enough*" improvements.**

  **Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least**

$$\frac{2\varepsilon}{|V|} w(A,B)$$

**Claim: Upon termination, the big-improvement-flip algorithm returns a cut $(A, B)$ so that**

$$(2+\varepsilon)\, w(A, B) \geq w(A^*, B^*)$$

- **May NOT in polynomial time**

  ☝ **stop the algorithm when there are no "*big enough*" improvements.**

  **Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least**

$$\frac{2\varepsilon}{|V|} w(A,B)$$

**Claim: Upon termination, the big-improvement-flip algorithm returns a cut $(A, B)$ so that**

$$(2+\varepsilon)\, w(A,B) \geq w(A^*, B^*)$$

**Claim: The big-improvement-flip algorithm terminates after at most $O(n/\varepsilon \log W)$ flips.**

# Maximum cut: big improvement flips

Local search. Within a factor of 2 for MAX-CUT, but not poly-time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least $\frac{2\varepsilon}{n} w(A, B)$

# Maximum cut: big improvement flips

Local search. Within a factor of 2 for MAX-CUT, but not poly-time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least $\frac{2\varepsilon}{n} w(A, B)$

Claim. Upon termination, big-improvement-flip algorithm returns a cut $(A, B)$ such that $(2 + \varepsilon) w(A, B) \geq w(A^*, B^*)$.

Pf idea. Add $\frac{2\varepsilon}{n} w(A, B)$ to each inequality in original proof. ▪

# Maximum cut: big improvement flips

Local search. Within a factor of 2 for MAX-CUT, but not poly-time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least $\frac{2\varepsilon}{n} w(A, B)$

Claim. Upon termination, big-improvement-flip algorithm returns a cut $(A, B)$ such that $(2 + \varepsilon)\, w(A, B) \geq w(A^*, B^*)$.

Pf idea. Add $\frac{2\varepsilon}{n} w(A, B)$ to each inequality in original proof. ▪

Claim. Big-improvement-flip algorithm terminates after $O(\varepsilon^{-1} n \log W)$ flips, where $W = \Sigma_e w_e$.

- Each flip improves cut value by at least a factor of $(1 + \varepsilon / n)$.
- After $n / \varepsilon$ iterations the cut value improves by a factor of 2.
- Cut value can be doubled at most $\log_2 W$ times. ▪

if $x \geq 1$, $(1 + 1/x)^x \geq 2$

# Neighbor relations for max cut

1-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in exactly one node.

k-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in at most $k$ nodes.

Can lead to better local optimum with larger range of local search.
But the computational cost increases exponentially w.r.t. $k$

# Neighbor relations for max cut

1-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in exactly one node.

k-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in at most $k$ nodes.

Can lead to better local optimum with larger range of local search.
But the computational cost increases exponentially w.r.t. $k$

KL-neighborhood. [Kernighan-Lin 1970]

- To form neighborhood of $(A, B)$:
  - Iteration $1$: flip node from $(A, B)$ that results in best cut value $(A_1, B_1)$, and mark that node.
  - Iteration $i$: flip node from $(A_{i-1}, B_{i-1})$ that results in best cut value $(A_i, B_i)$ among all nodes not yet marked.

cut value of $(A_1, B_1)$ may be worse than (A, B)

# Neighbor relations for max cut

1-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in exactly one node.

k-flip neighborhood. Cuts $(A, B)$ and $(A', B')$ differ in at most $k$ nodes.

Can lead to better local optimum with larger range of local search.
But the computational cost increases exponentially w.r.t. $k$

KL-neighborhood. [Kernighan-Lin 1970]

cut value of $(A_1, B_1)$ may be worse than $(A, B)$

- To form neighborhood of $(A, B)$:
  - Iteration $1$: flip node from $(A, B)$ that results in best cut value $(A_1, B_1)$, and mark that node.
  - Iteration $i$: flip node from $(A_{i-1}, B_{i-1})$ that results in best cut value $(A_i, B_i)$ among all nodes not yet marked.
- Neighborhood of $(A, B) = (A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$.
- Neighborhood includes some very long sequences of flips, but without the computational overhead of a $k$-flip neighborhood. $O(n^k) \rightarrow O(n^2)$
- Practice: powerful and useful framework.
- Theory: explain and understand its success in practice.

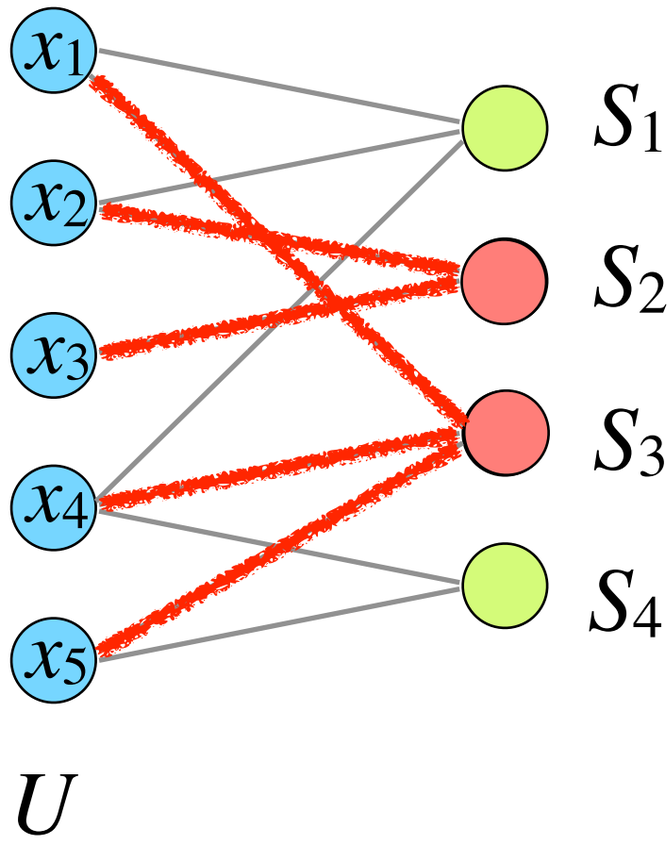
# Local Search

- Vertex cover
- Hopfield neural networks
- Max-cut
- Approximation algorithm for vertex cover
- Take-home messages

# Set Cover

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

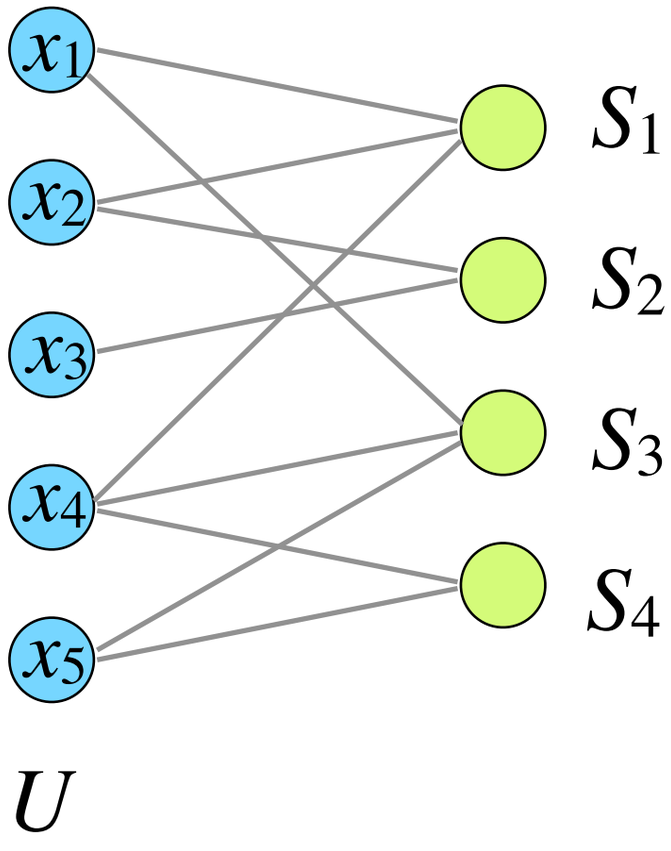
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



# Set Cover

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



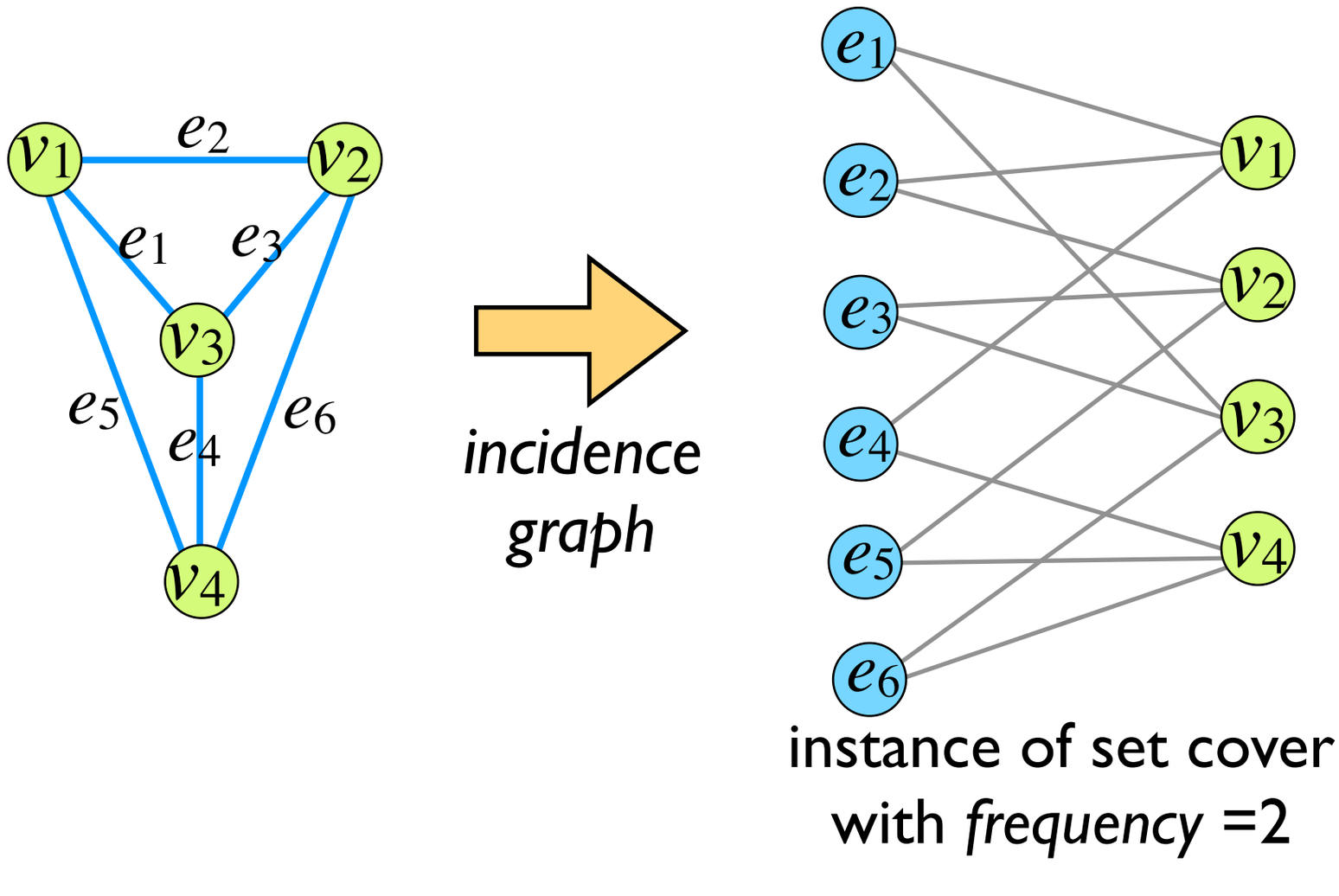

- **NP**-hard
- one of Karp's 21 **NP**-complete problems
- *frequency*: # of sets an element is in

$$frequency(x) = |\{S_i : x \in S_i\}|$$

# Vertex Cover

**Instance**: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in $C$.



# Vertex Cover

**Instance**: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in $C$.

- **NP**-hard
- one of Karp's 21 **NP**-complete problems

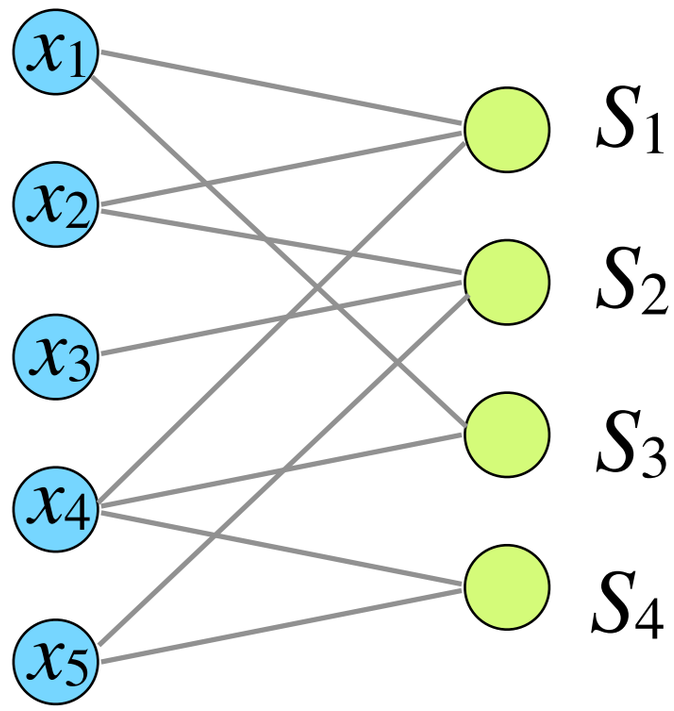
VC is **NP**-hard $\Rightarrow$ SC is **NP**-hard

Proved in Lecture 10.

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.

**Primal**: $C \subseteq \{1, 2, \ldots, m\}$ that $\cup_{i \in C} S_i = U$.

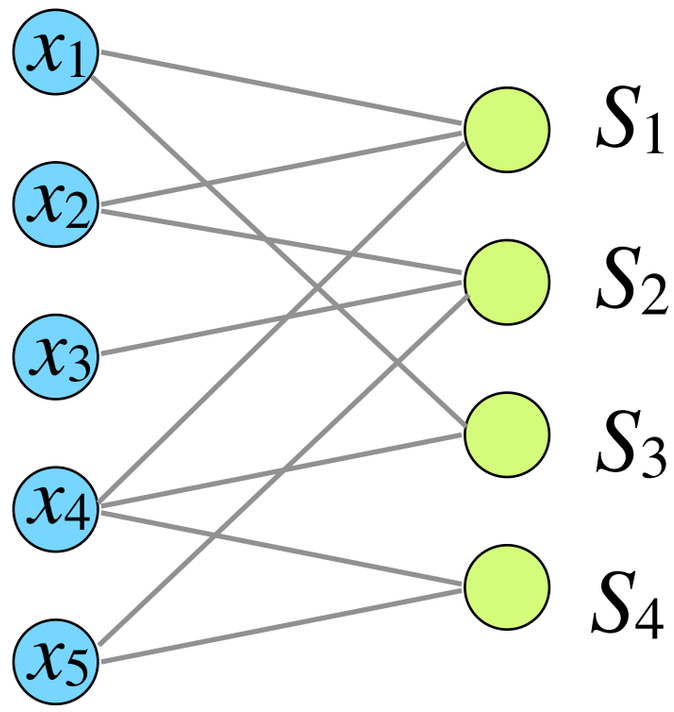
$\text{OPT}_{\text{primal}} = \min |\, C \,|$



**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

**Primal**: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$




**Dual**: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

$\forall C, \forall M$: $|\, M \,| \leq |\, C \,|$

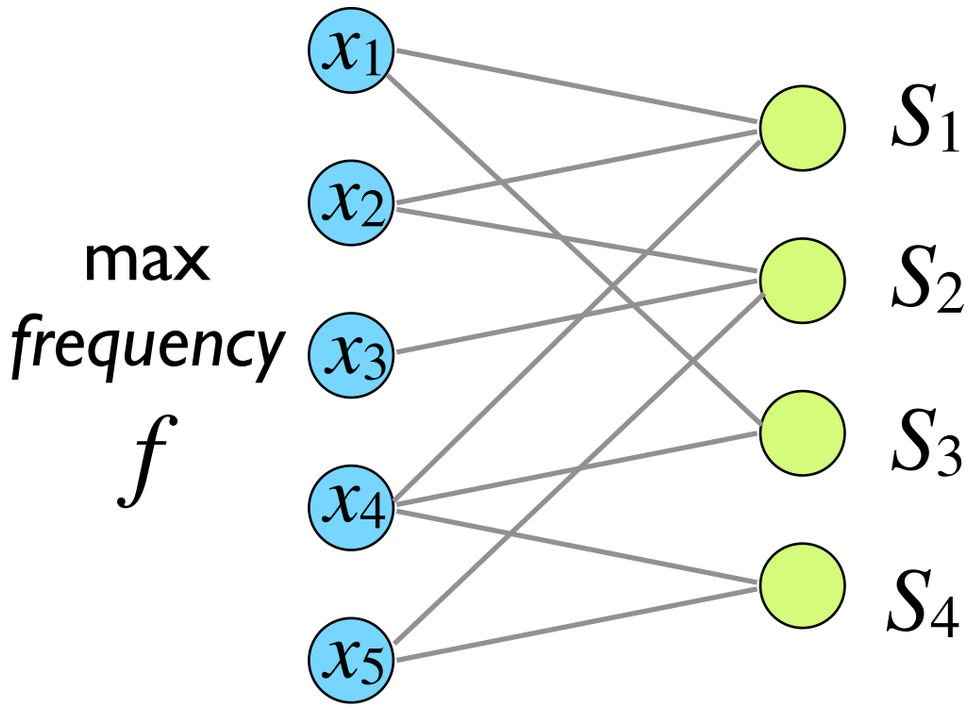
every $x \in M$ must consume a set to cover

$$\forall M: |\, M \,| \leq \text{OPT}_{\text{primal}}$$

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

**Primal**: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$



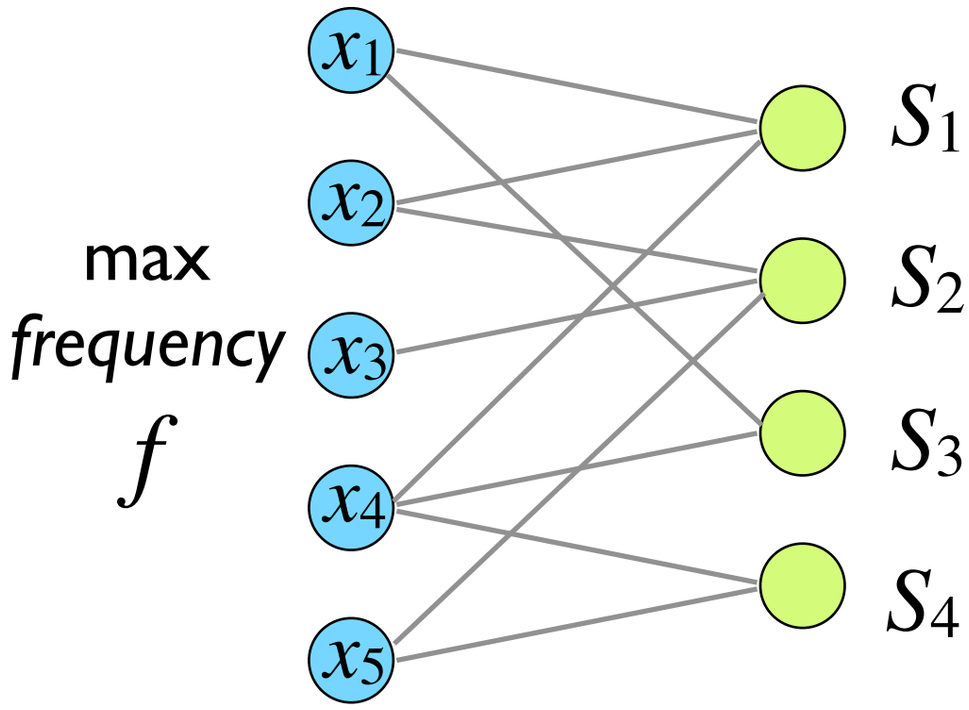

**Dual**: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

$$\forall M\text{: } |\, M \,| \leq \text{OPT}_{\text{primal}}$$

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

**Primal**: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$




max *frequency* $f$

Find a *maximal* $M$;
return $C = \{i : S_i \cap M \neq \emptyset\}$;

**Dual**: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

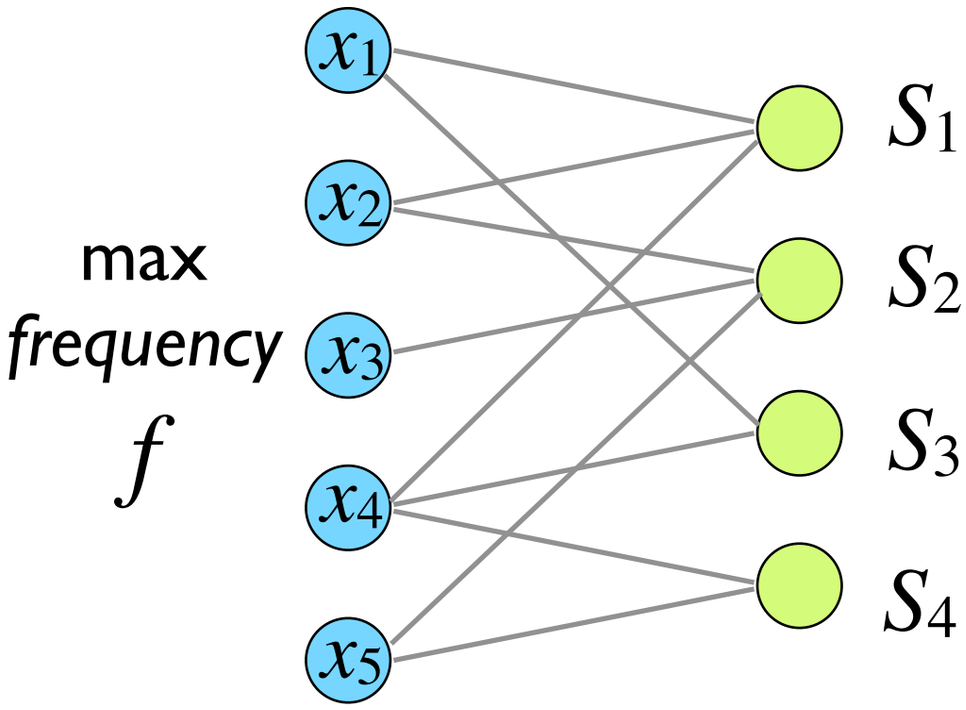
$$\forall M: \; |\, M \,| \leq \text{OPT}_{\text{primal}}$$

Maximal is not maximum:
Just choose an arbitrary order and add nodes in sequence until no node can be further included.

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

**Primal**: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$




Find a *maximal* $M$;
return $C = \{i : S_i \cap M \neq \emptyset\}$;

$M$ is *maximal* $\Rightarrow$ $C$ must be a cover

$$|C| \leq f \cdot |M| \leq f \cdot \text{OPT}_{\text{primal}}$$

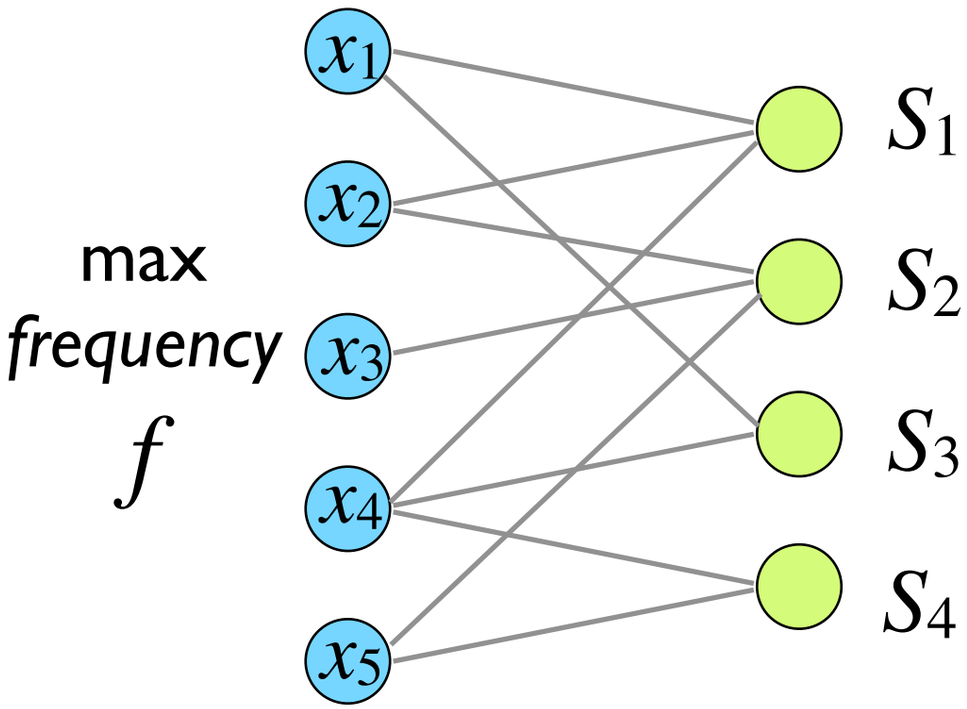
**Dual**: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

$$\forall M: \; |\, M \,| \leq \text{OPT}_{\text{primal}}$$

Maximal is not maximum:
Just choose an arbitrary order and add nodes in sequence until no node can be further included.

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.

Find a *maximal* $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$;
return $C = \{i : S_i \cap M \neq \emptyset\}$;




Frequency assumption:

$$\forall x \in U,\ |\{i: x \in S_i\}| \leq f$$

$$|C| \leq f \cdot \mathrm{OPT}$$

For vertex cover: This gives a 2-approximation algorithm.

# Vertex Cover

**Instance**: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in $C$.

a 2-approximation algorithm:

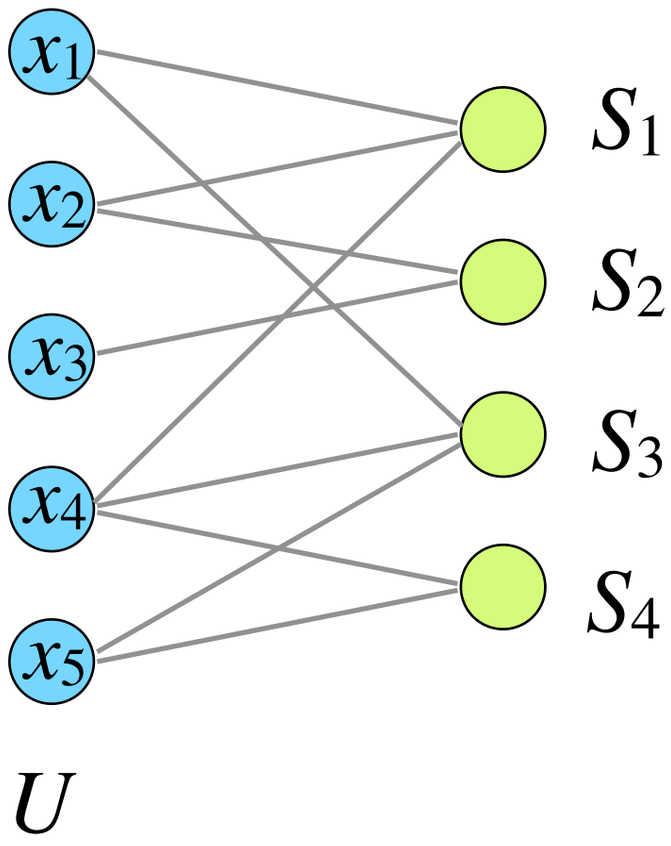
Find a *maximal matching*;
return the *matched* vertices;

- [Dinur, Safra 2005] There is no poly-time <1.36-approximation algorithm unless **NP = P**.
- [Khot, Regev 2008] Assuming the unique games conjecture, there is no poly-time (2-ε)-approximation algorithm.

# Set Cover

**Instance**: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

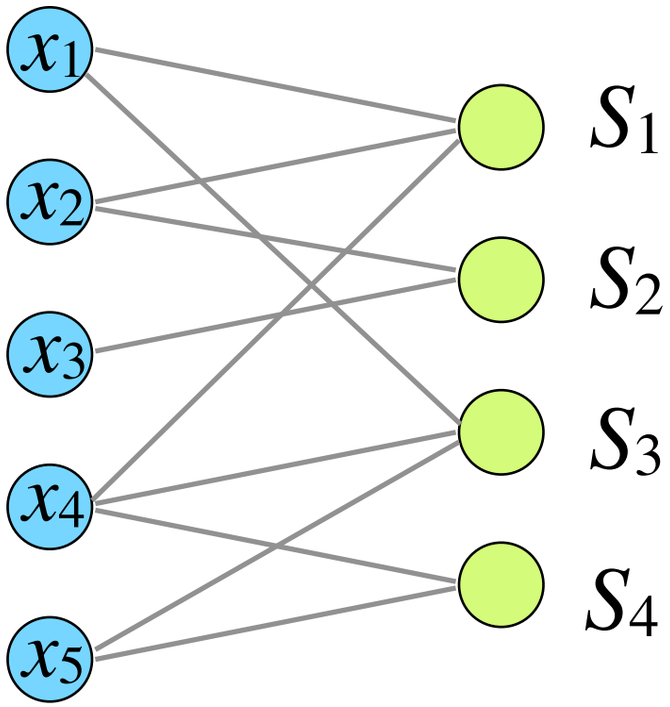
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.




**GreedyCover**

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add $i$ with largest $|S_i \cap U|$ to $C$;
  $U = U \setminus S_i$;

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.




**GreedyCover**

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add $i$ with largest $|S_i \cap U|$ to $C$;
  $U = U \setminus S_i$;

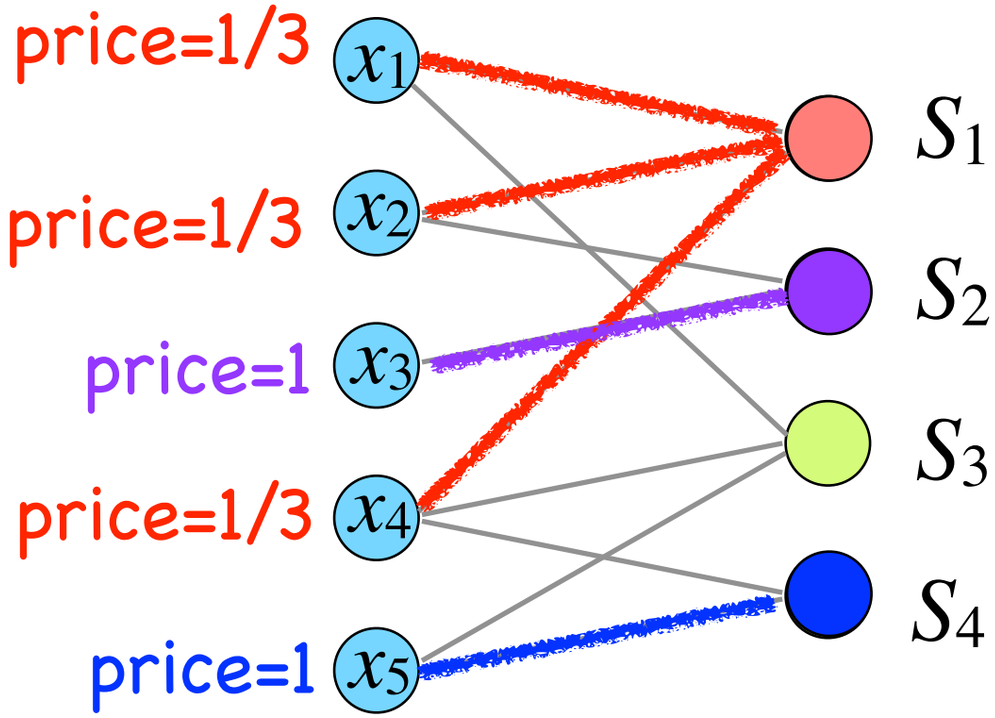
OPT($I$): value of minimum set cover of instance $I$

SOL($I$): value of the set cover returned by the **GreedyCover** algorithm on instance $I$

**GreedyCover** has *approximation ratio* $\alpha$ if

$$\forall \text{ instance } I, \quad \frac{\mathrm{SOL}(I)}{\mathrm{OPT}(I)} \leq \alpha$$

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.




**GreedyCover**

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add $i$ with largest $|S_i \cap U|$ to $C$;
  $U = U \setminus S_i$; ⌜∀x∈Sᵢ∩U, price(x)=1/|Sᵢ∩U|

$$|C| = \sum_{x \in U} price(x)$$

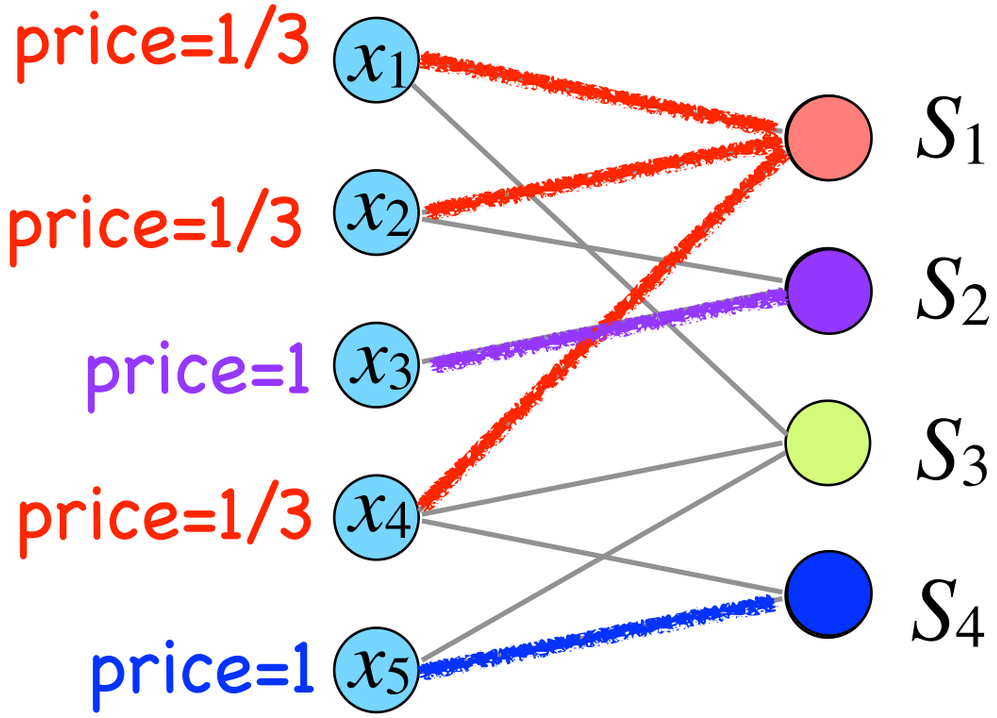
enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

elements can be *matched* to the sets in OPT cover

$$\exists S_i, \ |S_i| \geq \frac{|U|}{OPT} \Rightarrow price(x_1) \leq \frac{OPT}{|U|}$$

See https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.




**GreedyCover**

Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add $i$ with largest $|S_i \cap U|$ to $C$;
  $U = U \setminus S_i$; ∀x∈$S_i$, price(x)=1/$|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x)$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

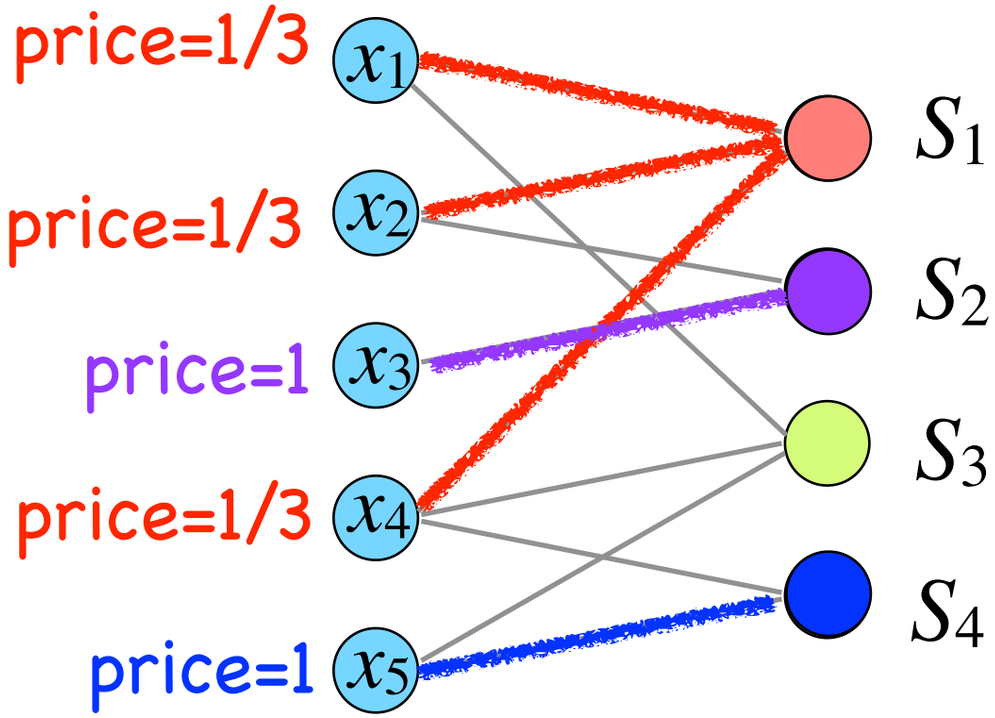
$$price(x_k) \leq \frac{OPT}{n-k+1}$$

consider $U_t$ in iteration $t$ where $x_k$ is covered:

$|U_t| \geq n\text{-}k\text{+}1$

all $S_i \cap U_t$ form a set cover instance: ≤OPT

**Instance**: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.




**GreedyCover**

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add $i$ with largest $|S_i \cap U|$ to $C$;
  $U = U \setminus S_i$; ∀x∈$S_i$, price(x)=1/$|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = H_n \cdot OPT$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

$$price(x_k) \leq \frac{OPT}{n-k+1}$$

**GreedyCover**

Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add $i$ with largest $|S_i\cap U|$ to $C$;
  $U = U \setminus S_i$;

- *GreedyCover* has approximation ratio $H_n \approx \ln n + O(1)$.
- [Lund, Yannakakis 1994; Feige 1998] There is no poly-time (1-o(1))ln $n$-approx. algorithm unless **NP** = quasi-poly-time.
- [Ras, Safra 1997] For some $c$ there is no poly-time $c$ ln $n$-approximation algorithm unless **NP = P**.
- [Dinur, Steuer 2014] There is no poly-time (1-o(1))ln $n$-approximation algorithm unless **NP = P**.

# Outline: Local Search

- Vertex cover
- Hopfield neural networks
- Max-cut
- Take-home messages

# Take-Home Messages

- Local search: modifying the state of the solution to its neighbor state, and check whether improvement appears.

- Usually finding local optimal solution instead of global optimum.

- It is tricky to define neighbor states and escape from local optimums.

- Further reading: the scheduling problem discussed in:

  - https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

  - https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

Thanks for your attention!
Discussions?

# Reference

Algorithm design: Chap. 12.

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani: Sec. 9.3

Mackay book Chap. 42 on Hopfield nets: https://www.inference.org.uk/itprnn/book.pdf

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/12LocalSearch.pdf

Yitong Yin's lecture:

https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf